AFRL-RI-RS-TR-2016-101

STATIC ANALYSIS OF NUMERICAL ALGORITHMS

KESTREL TECHNOLOGY, LLC

APRIL 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-101 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM MCKEEVER
Work Unit Manager

/ S /
RICHARD MICHALAK
Acting Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
APRIL 2016	FINAL TECHNICAL REPORT	NOV 2013 – NOV 2015

4. TITLE AND SUBTITLE

STATIC ANALYSIS OF NUMERICAL ALGORITHMS

5a. CONTRACT NUMBER
FA8750-14-C-0009

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
63781D

6. AUTHOR(S)

Matthew Barry, Eric Bush, Doug Smith,Devesh Bhatt, David Oglesby, Anca Browne, Steve Hickman

5d. PROJECT NUMBER
ASET

5e. TASK NUMBER
13

5f. WORK UNIT NUMBER
KT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Kestrel Technology LLC
3260 Hillview Ave
Palo Alto CA 94304-1225

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-101

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2016-1710
Date Cleared: 31 MAR 2016

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This document reports on a two-year research project by Kestrel Technology and Honeywell Aerospace Advanced Technology to combine model-based development of complex avionics control software with static analysis of the generated code to achieve assurance levels not available from either technique practiced separately. We concentrated on two classes of numerical algorithms, linear digital filters and integrating accumulators, modifying existing versions of Honeywell's HiLiTE model-based development system and Kestrel's CodeHawk abstract interpretation system to share domain specific information about implementations of these algorithms. This allowed CodeHawk to exploit model-level specifications and theoretical input bounds from HiLiTE concerning the generated C code, producing a much more precise over-approximation of the output bounds and accumulated floating-point error bounds than would be possible with generic abstract interpretation techniques. These static analysis results were then fed back into HiLiTE to be further exploited in the formal verification of the generated code.

15. SUBJECT TERMS
Model-based development, abstract interpretation, linear digital filter, integrating accumulator, recurrence relation, floating-point error bounds, numerical algorithms, software verification, formal methods

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM McKEEVER**
U	U	U	SAR	73	19b. TELEPHONE NUMBER (*Include area code*) **(315) 330-2897**

TABLE OF CONTENTS

LIST OF FIGURES

1. SUMMARY

This document reports on a two-year research project by Kestrel Technology and Honeywell Aerospace Advanced Technology to combine model-based development of complex avionics control software with static analysis of the generated code to achieve assurance levels not available from either technique practiced separately. We concentrated on two classes of numerical algorithms, linear digital filters (LDFs) and integrating accumulators, modifying existing versions of Honeywell's HiLiTE model-based development system and Kestrel's CodeHawk abstract interpretation system to share domain specific information about implementations of these algorithms. This allowed CodeHawk to exploit model-level specifications and theoretical input bounds from HiLiTE concerning the generated C code, producing a much more precise over-approximation of the output bounds and accumulated floating-point error bounds than would be possible with generic abstract interpretation techniques. These static analysis results were then fed back into HiLiTE to be further exploited in the formal verification of the generated code.

We made some unexpected discoveries during the first year analysis of LDFs concerning analytic solutions to output and error bounds for filters that changed the architecture of our first year approach, and redefined our focus for the second year effort. In particular, Anca Browne's discovery of an analytic technique for computing filter bounds obviated the need for the iterative symbolic evaluation of classical abstract interpretation, making the time to analyze a filter negligible. This contrasts with comparable filter analyses performed for Airbus that take on the order of 20 hours to get less precise results. Extending this technique to floating point error analysis, she discovered how to generate a closed-form solution for the error as a function of the number of iterations of the recurrence that are performed. This resulted in CodeHawk's "error bounds" for filters being reported back to HiLiTE not as an instance-specific numeric interval, as was originally envisioned, but as parameters to a generalized error function, allowing HiLiTE to analyze multiple scenarios with different durations from a single analysis result. When these techniques were generalized to integrating accumulators in year 2, CodeHawk was able to report both output and error ranges as parameters to a generalized function.

We built specialized versions of both HiLiTE and CodeHawk to exploit the shared model-level information and exchange their results over a JSON file interchange architecture. This combined system was used first to experiment with the joint modeling and analysis techniques, and then to run a series of test suites for filters and integrating accumulators that explored the range of variety, and analysis results, among those algorithms. These test suite results are summarized in detail in Sections 4.2 and 4.3 below.

2. INTRODUCTION

The objective of this project was to develop a software engineering approach and tool set that combines the relative strengths of model-based and code-based analyses to provide a comprehensive and scalable analysis capability for numerical algorithms in complex systems software. Our approach was to utilize static analysis of numerical algorithms in combination with model-based design technology to yield considerable benefits compared to applying static analysis in isolation. The results of this effort are intended to help detect and mitigate a large class of defects that occur due to differences between the intended semantics of design models and the actual behavior of the software.

Large and complex systems control software increasingly is being developed using model-based development tools. In this approach a design model of control algorithms is constructed using control function blocks in a data flow notation. The transfer function semantics is available at the model level along with the semantics of feedback loops, periodic rates, and design patterns such as counters, validity status of signals, reset semantics, mid-value selection, etc. Some model-based toolsets perform extensive analysis of models for early detection of several types of design defects and have been used as DO-178B qualified tools in certification of large scale commercial avionics systems. These tools can reason with full semantics of control transfer functions and discrete constructs, but the bounds on the ranges and errors can be conservative and cannot take into account the actual source code constructs and numeric operations and the artifacts introduced by the translation of the design model into source code and the impact of object code execution on the processing architecture. This challenge is commonly faced when abstractions, assumptions, and restrictions are utilized to estimate the behavior of object code at the design model level.

Abstract interpretation is a fully automated mathematical process for discovering properties of the execution behavior of a program. Because they derive from a formal representation of a program's behaviors, the results of this kind of static analysis have the status of mathematical proofs. Informally, abstract interpretation operates by computing an envelope of all possible values of the variables at each point of the program. For example, the possible values of scalar variables can be represented by a collection of intervals (one for each variable) or a convex polyhedron (each dimension of the affine space representing a program variable). These envelopes are then employed as lemmas to prove safety properties of the program, like the absence of arithmetic overflows or out-of-bounds array accesses. Some abstractions yield more accurate envelopes, which allow more safety conditions to be discharged, but may require higher computational times. For example, using intervals to compute the range of variables is extremely fast and they can be applied to analyze million lines of code. Using convex polyhedra usually yields much tighter bounds, but the exponential complexity of the underlying algorithms makes this approach impractical for more than a few hundred lines of code. The complexity of engineering a static analyzer that can analyze real codes effectively lies in the combination of different levels of abstraction, so that precise and costly abstractions (like convex polyhedra) are only applied to those parts of the program that require it, whereas rougher but faster abstractions are employed elsewhere. There is no automated way of doing this combination, which is why

industrial-strength static analyzers are usually specialized for a particular application or family of applications.

Our specific approach in this project was to specialize Kestrel Technology's CodeHawk abstract interpretation technology to particular classes of numerical algorithms in avionics systems control software generated by Honeywell's HiLiTE model-based development software. This marriage of a model-level and a code-level tools allows both the dual verification of model and C code correspondences, as well as considerable improvements to the precision of CodeHawk's abstract interpretation due to its exploitation of domain-specific, model-level information from HiLiTE. We built a custom version of both tools to exploit a round-trip JSON integration from model-analysis to code, to code-analysis informed by model-analysis, and back to model-analysis confirmed by code-analysis.

2.1. Original SOW Vision

The original SOW envisioned specializing this combined model/code analysis architecture to a specific class of numerical algorithms, Linear Digital Filters (LDFs), in the first year, then using the results of this experience to explore other classes of numerical algorithms over a specific abstract domain, zonotopes, in the second year. Our first year exploration of LDFs, however, led to some unexpected discoveries about the abstract interpretation of filters, when the abstract interpreter knows (via model-level input) that the C code it is analyzing actually implements a filter function. This led to a radical redesign of the analyzer, and a shift in focus for the abstract domains first envisioned in the SOW. This re-orientation also led us to re-conceptualize the second year effort toward a particular class of numerical algorithms (integrating accumulators), instead of an abstract interpretation domain (zonotopes).

2.2. Discoveries about Abstract Interpretation of Filters

Abstract interpretation attempts to compute an approximation of a program's behaviors over all possible execution paths, without any actual inputs, using only the semantics of the programming language to inform its analysis. Without knowledge of inputs, or of what algorithms the program is executing, the inferred behaviors will necessarily be a conservative over-approximation of the actual behaviors. This over-approximation can often be used to prove properties about the actual behaviors of the program, however, if the properties are also true of the over-approximation of those behaviors. We recast this problem for LDFs by using HiLiTE to 1) inform CodeHawk that the C code it is analyzing implements a LDF, and 2) provide theoretical bounds on the values of all inputs to the filter. CodeHawk is then able to compute an approximation of actual bounds of the output values that is much more precise that would be possible without these hints.

2.2.1. Filter-specific widening

Filters are a special class of iterating numerical algorithms that use the results of prior iteration steps to constrain the computation of the next step. This leads to a natural tamping down of the input variety and causes the bounds of possible outputs to converge to a smaller range. Without this special knowledge, uninformed abstract interpretation will continue to "widen" the iterated values in search of a much more conservative convergence. The challenge for abstract interpretation of filters is in developing a special widening analysis for filters that takes account of the unique cancellation effects that one output has on the next input.

We originally anticipated that our filter-specific analyzer would be specialized to particular *classes* of filters (exploiting a filter-class parameter from HiLiTE) so that widening could exploit the unique cancellation characteristics of these classes. We discovered, however, that this class-specialization is unnecessary. We developed an incremental widening algorithm, taking cancellation into account, which appears to work well for all linear filters. Moreover, we discovered that a stable bound on the filter output can be computed symbolically, via closed form solution, thus obviating the need for the traditional iterative widening of classical abstract interpretation.

2.2.2. Floating Point Error Bounds

Another original goal of this project was to take account of floating-point round-off errors when real-numbered models are implemented as floating-point code. Theoretical bounds on filter outputs derived from models over the real numbers will often not anticipate possible round-off effects in the generated C code, which will vary with machine precision. Abstract interpretation typically performs its analysis over the infinite precision reals as well, so a combination of model-level and code-level analyzers might still miss this important delta.

We had originally planned to run the analysis for each filter against both CodeHawk's native real/rational abstract domain and a newly implemented floating-point abstract domain, and then compare the output range approximations to compute an approximation of the round-off error exposure for floats. Our initial implementation of this comparison revealed that the accumulated floating-point error range is a function of the number of sampling periods for the filter and thus could be computed independently. When this float-error accumulation is combined with the cancellation effects of subsequent inputs, there is an inherent stabilization of the error accumulation. This result turned out to be surprising in that the Honeywell team had estimated error accumulations to be in the 1%-10% range per 1,000,000 samplings. The analysis results yielded error ranges that were orders of magnitude lower.

It appears that filters' inherent functionality in stabilizing sensor output works to stabilize, rather than accumulate, round-off errors. We were further able to derive an optimal function to compute the error-accumulation for a filter-instance/sampling-period combination, so rather than reporting error bounds, CodeHawk now reports the constant terms to this function, specific to the filter instance (input bounds) under analysis, rather than an instance-specific range. The Honeywell team can now experiment with various sampling periods for this filter instance off-line without having to rerun the CodeHawk analysis.

2.3. Re-orientation of Year 2

The original idea in the SOW for the second year of this project was to extend the LDF analysis to an interestingly different class of numerical algorithms. This was originally characterized, not by an algorithm class (like LDFs), but by a new analysis domain: zonotopes. But our first year experience concluded that this work would be more valuable to the aerospace community if the focus was on some particular numerical algorithm class that has practical interest for aerospace, rather than on a particular abstract interpretation domain. Our somewhat surprising discoveries about floating-point error convergence in LDFs led to an interest in the converse class of numerical algorithms where such errors do accumulate, often undetected at design time. We started with an "accumulating integrator" class of algorithms that was the source of a floating-

point error design flaw in the Patriot Missile control software that failed to detect an incoming Iraqi Scud, allowing it hit the Army barracks in Dhahran, Saudi Arabia in 1991 (killing 28 Americans). If there had been an abstract interpreter, specialized to this algorithm class over the "arithmetic-geometric progression" domain, the floating-point round-off error could have been discovered at design/implementation time, before the software was deployed.

We began with a particular case study: the Patriot Missile failure, then generalized to other numerical algorithms in this general class. The new abstract domain is the arithmetic-geometric progression domain. It is useful for analyzing the accumulation of round-off errors in algorithms that, unlike filters, accumulate rather than stabilize their errors. We got a head start on the analysis of this class of algorithms through characteristics that they share with the analytic solutions discovered for LDFs. Although the floating-point errors accumulate for this class (leading to application failures if the bounds are not correctly anticipated), rather than converge as they do for LDFs, the error-bounds can still be characterized by a single, analytic function over the number of iterations of the accumulator. This allowed us to report the error-bounds output for accumulators in the same format as for our LDF work. We further discovered that the *range* bounds on the accumulator output values could be more generally represented as coefficients to an analytic range function, rather than a specific numeric interval, so the code-analysis results passed from CodeHawk back to HiLiTE are more general for accumulators than for LDFs.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

The goal of this project was to build a combined system of model-level analysis and code-level analysis that would allow each analyzer to exploit artifacts from the other. Since we began with two existing, unrelated model analysis and code analysis systems, the Statement of Work for the project was organized around three major subtasks: 1) enhancements to Kestrel Technology's CodeHawk abstract interpretation software to exploit model-level inputs, 2) enhancements to Honeywell's HiLiTE modeling and code generation software to generate those inputs and exploit the resulting analysis, and 3) an integration architecture for allowing the enhanced versions of both tools to work together as a single tool, presenting a unified analysis to the end user. The following three subsections discuss the starting methods, assumptions, and procedures for each of these subtasks, respectively.

3.1. Control Algorithm Capture in Models and Analysis / Code Verification

Use of Model-Based Development (MBD) techniques for software development for control algorithms has become a common practice in avionics for systems of small to large complexity. In this approach, a design model of control algorithms is constructed using control function blocks in a data flow notation. The control function blocks include arithmetic operators, digital filters, integrators, timers, limiters, etc. that are composed in various patterns such as proportional-integrator (PI) controllers that utilize various aspects of control including feedback error compensation, integrator anti-windup logic, and shaping the characteristics of the inputs from sensors/plant and the command to actuators to meet the control requirements. Besides the control-related constructs, the controller software also includes discrete, time-dependent logic including timers and counters for implementing mode and states changes.

The controller design is captured in design models using tools such as MATLAB Simulink and SCADE. Source code in C-language is generated from these models that is then used as the airborne code in avionics systems. Processes and automation tools have been deployed to certify these systems to the DO-178B (and the revised DO-178C) standard. The processes follow the verification objectives defined in DO-178C that start with the software high-level requirements being refined into design models from which source code is generated that gets compiled and linked into airborne executable object code. Figure 1 illustrates this process. Note that there are several verification objectives defined in DO-178C for the development artifacts – only a few of the objectives are shown in this figure for the design models and the source code.

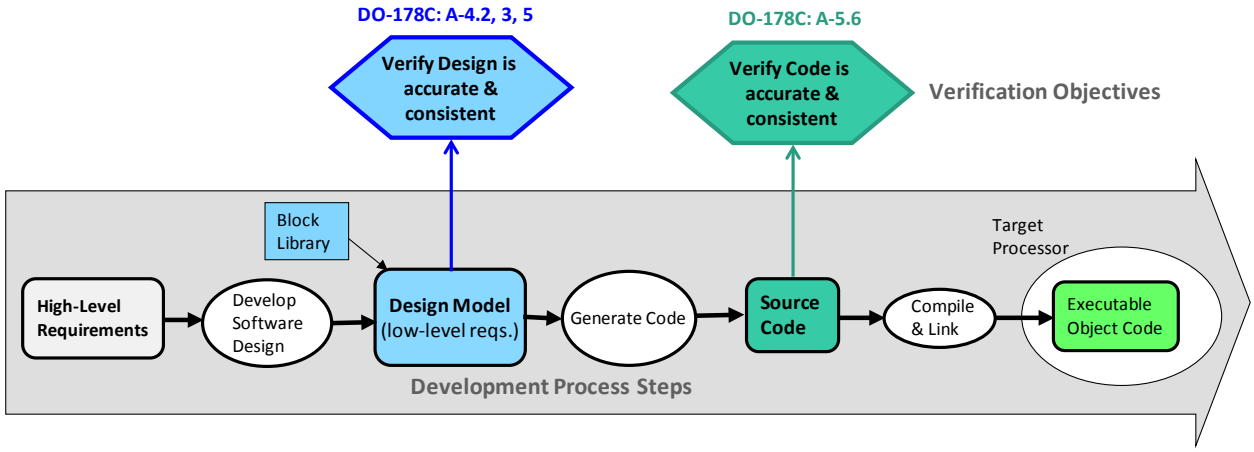

Figure 1. Software Development Process and Verification Objectives

The objective shown in Figure 1 for the design model is to verify that the low-level requirements (i.e., the elements of the design model) are accurate, consistent, and verifiable. This generally includes absence of overflow (e.g., division by zero), untestable conditions, and unreachable model elements. The source code verification objectives are also quite similar, with the addition of memory safety specific to source code constructs, memory usage limits, etc. A point to note here that these verification objectives are quite similar and are applied independently at the design model level and source code level – with different methods and analysis tools at these two levels to perform the verification.

Need to achieve precise and consistent verification results across model-level and code-level analysis: At the core of the static analysis techniques at the model or code level is the derivation of conservative range and numerical error bounds on the variables in the algorithm – using these bounds various properties can be proven. In complex numerical algorithms, this is a hard problem. For example, for transfer functions such as linear digital filters, there is a recurrence relation of the output of the filter to the previous step's output and input values, which in turn depend upon the values in the step before that. It turns out that general-purpose static analysis techniques give poor results on linear digital filters. What is needed are specialized techniques to derive sound and precise results. The application of such specialized techniques needs to draw upon the semantics of the modeling constructs so that the tools can identify which parts of the design or code a particular specialized analysis needs to be applied and what are the parameters of interest in the particular instance of usage. This is a common practice in use of formal analysis and model checking tools – providing auxiliary lemmas and parts of proof to the tool to make the problem tractable.

A problem observed with the current state of the model-level and code-level analysis is that the users get different answers from the tools applied at these two levels due the practical limitations of the analysis capabilities of the tools. For example, a control-theory based analysis may predict reasonable range bounds at the output of a filter but a code analysis tool may derive very large bounds for the filter since that tool does not apply a specialized analysis technique for this code. This inconsistency (and large bounds reported by the code analysis) creates a problem for the user to understand and resolve the discrepancy. Furthermore, large bounds can lead to false

alarms and trigger unnecessary additional work steps in the certification process to manually analyze and resolve each situation.

The research goal of this project is to investigate/develop a model-based software engineering approach and tool set that combines the relative strengths of model-based and code-based analyses to provide a comprehensive and scalable analysis capability for numerical algorithms in complex systems software.

3.1.1. Use of Semantics at the Design Model Level to Aid Analysis

Models use control transfer functions such as filters, integrators, gains to build a control algorithm. Figure 2 shows a simple Simulink model that is part of a control algorithm feedback loop – specifically compensation of the feedback error from the plant sensor to meet the control objectives of rise time and overshoot. There are two lead-lag filters in series; instances of the generic lead-lack filter block with specific values of the time-constant parameters.

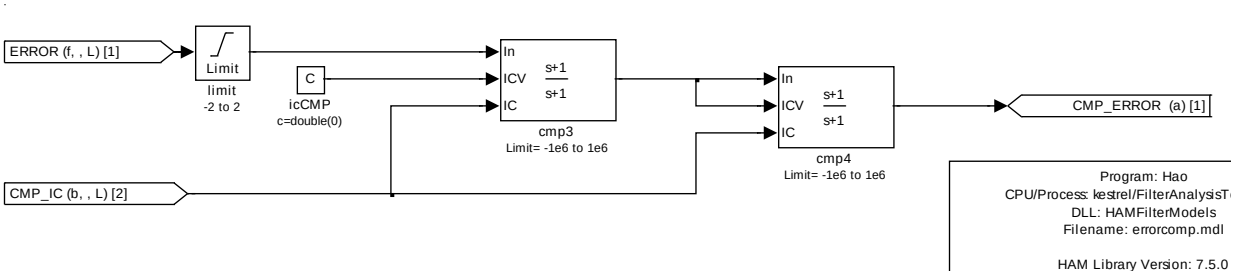

Figure 2. Example Model of Control Algorithm using Lead-Lag Filters

The transfer function of the lead lag filter is shown in Figure 3. The control algorithm analysis and simulation in closed-loop mode starts with the continuous domain Laplace transform. During the model-based development process, that is translated to the transfer function in the discrete domain (Z-domain) that is implemented in the controller software. This is a necessary step since the controller software is implemented as a periodic thread where the transfer function is evaluated at each periodic step (frame). From the Z-domain transform, we derive a difference equation that is then implemented in the software. The terminology used in Figure 3 is shown on the right hand side: *u* is the input and *y* is the output of the filter. The subscripts *n*, *n-1*, *n+1* denote the relative number of a particular time step in the sequence.

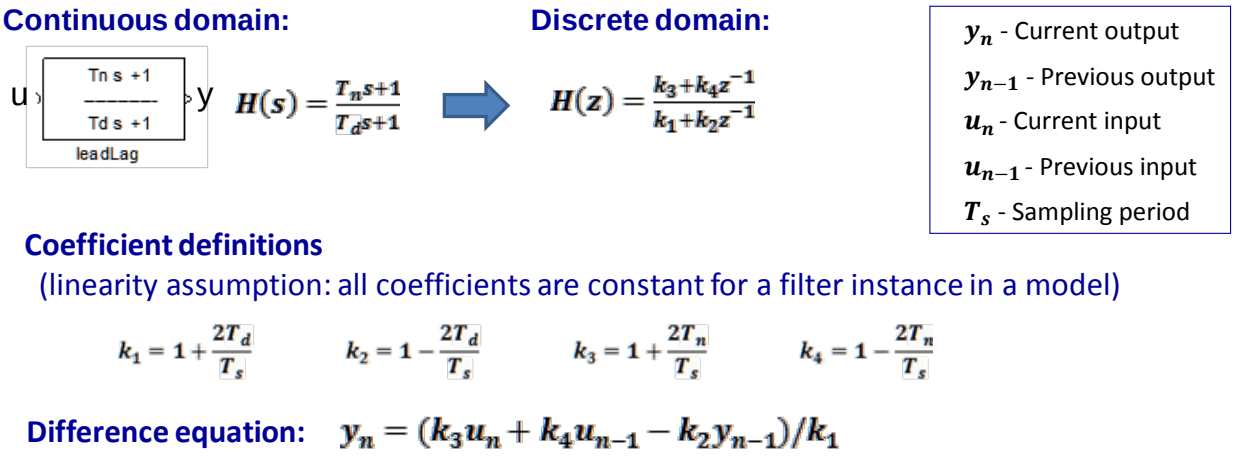

Figure 3.Transfer Function for a Lead Lag Filter and Derivation of Difference Equation

Using the Z-domain transfer function *H(z)*, a difference equation can be derived as shown in the figure. The difference equation computes the value of *y* in the current step (y_n) in terms of output value in the previous step and input values in the previous and current steps. The coefficients *k1* to *k4* are defined in terms of the sampling period *Ts* and the filter time constants: *Tn* is the numerator time constant and *Td* the denominator time constant of the transfer function. Note that the terms u_i and y_i in the difference equation correspond to variables in the code where u_{n-1} and y_{n-1} are state variables that hold the values of *u* and *y* from the previous step. The difference equation and the form of its corresponding translation to code are very important from the perspective of the code-level analysis that is described in another section in this document.

3.1.2. Exploration of Model-Based Code Generation Approach

Specific model-level encapsulation guidelines and code generation options are required to ensure the generated code will not exhibit intermingling of the numerical algorithm's code with the code from other parts of the model. This presents an unnecessary difficulty for the code-level analysis since the intermingled code cannot be cleanly mapped to the appropriate abstract domain corresponding to the algorithm characteristics.

Using MATLAB Simulink, we performed experiments with different options of modeling abstractions and code generation and selected a strategy that is best suited for integrated static analysis of models and code. For each filter block, we construct a reference model for simulation of its discrete behavior and code generation. The reference model is built with basic arithmetic blocks and unit delay blocks in Simulink.

From the continuous transfer function of a filter block, we first derive its discrete transfer function with parameters expressed in the continuous domain. The correctness of the discrete transfer function can be verified by the difference equation converted from it. There are several methods to transform continuous transfer functions to discrete and vice versa. We use the *bilinear transform* which essentially substitutes "***s***" in the continuous transfer function using the first order approximation:

$$s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}} \quad (1)$$

.

For example, from the continuous transfer function *H(s)* of the lead-lag filter, we have

$$\boldsymbol{H(s)=\frac{T_n s+1}{T_d s+1}} \Rightarrow \boldsymbol{H(z)=\frac{k_3+k_4 z^{-1}}{k_1+k_2 z^{-1}}},\ \boldsymbol{k_1=1+\frac{2T_d}{T_s}},\ \boldsymbol{k_2=1-\frac{2T_d}{T_s}},\ \boldsymbol{k_3=1+\frac{2T_n}{T_s}},\ \boldsymbol{k_4=1-\frac{2T_n}{T_s}}. \qquad (2)$$

Hence the difference equation is $\boldsymbol{y_n=(k_3u_n+k_4u_{n-1}-k_2y_{n-1})/k_1}$. Note that $\boldsymbol{T_n}$ and $\boldsymbol{T_d}$ are time constants defined in the application model for each filter instance similar to the way a constant block's value is specified. The sample time, $\boldsymbol{T_s}$, is linked to the application model's sample time by Simulink for all filter instances.

Figure 4 shows a Simulink model that can be referenced by application models. An application model employs a filter by connecting signals used in an application to a model reference that links to the implementation of the filter.

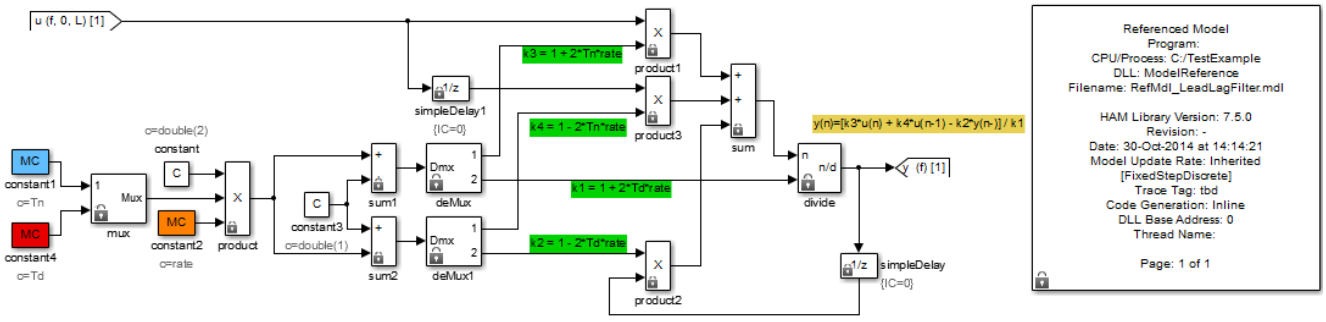

Figure 4. Reference model for lead-lag filter block

The reference model is constructed by the following steps:

1. Create a new Simulink model file.
2. Set up the reference model parameters as shown in Figure 5.
    a. Navigate to "View"-"Model Explorer" to open the Model Explorer window.
    b. Click "Model Workspace" in "Model Hierarchy" menu on the left side of the window.
    c. Click "Add Simulink parameter" button on the top tool bar.
    d. Double click the parameter to set its name/default_value/type/dimension…

e. Add all the parameters (separated by comma) into the model argument

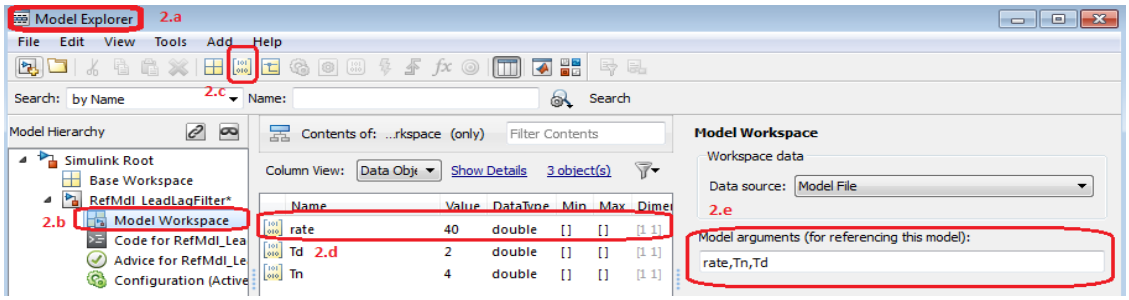

Figure 5. Setting reference model parameters

3. Create the Simulink block diagram according to the difference equation.
4. Assign constant blocks with corresponding parameters from workspace.

Figure 6 shows an application model with lead-lag filter blocks. The code for the lead-lag filter is generated as a separate function. The coupling between the main model's code and the filter function is achieved via the variables corresponding to the input and output of the filter function. Additional relevant parameters are specified when the model reference is instantiated, allowing the application model to define the time constants for each instance and enforcing a consistent same period between the application model and all filter instances.

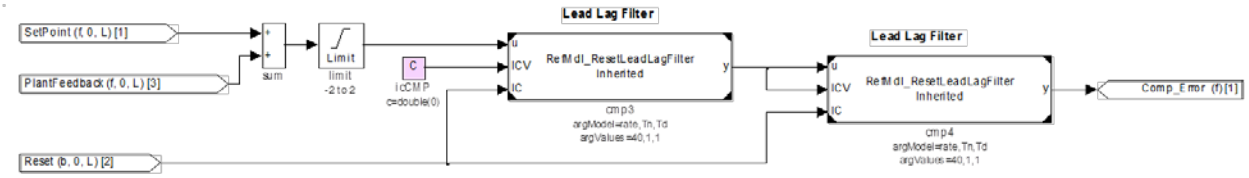

Figure 6. An Application Model Containing a Lead-Lag Filter Function

The code generated for the model is shown below. The input variables are `errorcomp_B.Limit_1`, `errorcomp_P.Constant_1_Value`, and `Reset`. The output variables is `rtb_cmp3_y`. There are two state variables, `errorcomp_DWork.cmp3_DWORK1.rtb` and `errorcomp_DWork.cmp3_DWORK1.rtdw` . The sample time, `errorcomp_P.cmp3_rtp_rate`, is automatically inherited from the application thread level. Finally the filter instance time constants, `errorcomp_P.cmp3_rtp_Tn` and `errorcomp_P.cmp3_rtp_Td`, are specified by the application model designer.

```
void errorcomp_step(void)
{
  …
  errorcomp_B.Limit_1 = saturate_dbl(errorcomp_B.Sum1,
                                     errorcomp_P.Limit_1_LowerSat,
                                     errorcomp_P.Limit_1_UpperSat);
  /* call to lead lag filter function for 1st filter instance */
  RefMdl_ResetLeadLagFilter_p(&errorcomp_B.Limit_1,
                              &errorcomp_P.Constant_1_Value, &Reset,
                              &rtb_cmp3_y,
                              &(errorcomp_DWork.cmp3_DWORK1.rtb),
                              &(errorcomp_DWork.cmp3_DWORK1.rtdw),
                              errorcomp_P.cmp3_rtp_rate,
```

```
                                errorcomp_P.cmp3_rtp_Tn,
                                errorcomp_P.cmp3_rtp_Td);
    …
}
```

It is important that the referenced model code closely reflects the difference equation for the filter function. Other implementations may incorporate coefficients into feedback loops to produce the same results with variable—rather than constant—coefficients that throw off code-level analysis.

Some filters (those named Variable<filter type>) take tau (or T_n or T_d) as an input signal. This would allow the model to vary tau continuously, which is not supported by these analyses. The coefficients need to be fixed for the analysis. In practice, however, tau is never varied continuously since that can make the entire control analysis non linear. In general, the tau input is a way to decouple the filter for more flexible models. For example, product line modeling reuses models for different variants of a component. The architecture is the same for all part numbers, but details change between parts. In such a scenario, a different tau is specified for different configurations, but it is fixed for each. In other usages, where multiple modes exists in the avionics subsystems, tau can change across mode changes, but is constant within a single mode. This allows separate piecewise analyses to be performed for each mode.

3.2. Abstract Interpretation

Given a model and code generated from the model, we want to provide mathematical evidence that the code faithfully implements the model. For linear digital filters (LDFs), the code implements a linear recurrence equation derived from a continuous-domain model. A filter takes a stream of inputs (e.g. a stream of sensor readings), and produces a stream of output values in accord with the linear recurrence. An accumulating integrator takes a stream of inputs and also produces a stream of outputs, again according to a linear recurrence, but the output values and their floating point errors can grow without bound with the number of iterations.

Two main analysis issues arise: (1) what is the range of possible output values considered over all possible input streams, and (2) what is the range of possible errors in the output stream due to the floating point implementation of real-valued variables in the continuous model?

Generally, static analysis reasons about all possible behaviors of a program. Since there are usually a very large (or even infinite) set of possible inputs, it is usually not feasible to reason about all the inputs individually. Instead, static analysis treats all inputs at once by characterizing their common structure, via type information and logical constraints. The goal is to symbolically simulate the execution of the program over this input characterization, yielding a characterization of all possible behaviors. Typically, programs have a very complex set of behaviors and so it is only feasible to generate an approximation, analogous to finding the hull of a complex set of points in space. If the approximation is an over-approximation (i.e. an upper bound), then any property that we can prove about the over-approximation also holds for all behaviors of the program.

The general theory of static analysis, called Abstract Interpretation, was developed in the 1970's by Cousot and Cousot [1]. The technique is widely used and can be efficient enough for commercial use. One of the success stories of abstract interpretation is the complete analysis of

the Airbus 380 flight control software for out-of-bound memory access and arithmetic exceptions [2]. CodeHawk is a state-of-the-art implementation of abstract interpretation.

The key idea of abstract interpretation is *abstract domains* that allow computing an over-approximation of the behaviors of a given program. An abstract domain provides (1) an abstract type to represent concrete program states, and (2) abstract functions to represent the effect of concrete state-changing actions. Rather than simulate the concrete program, abstract interpretation uses abstract domains to construct and simulate an abstract version of the program. The simulation is computationally cheaper in the abstract program because the abstraction throws away information. However the very act of throwing away information causes a loss of precision in the simulation/analysis process. Hence there is a tradeoff in abstract interpretation between computational complexity of the analysis and precision of the results.

To illustrate the loss of precision due to abstraction, suppose that we want to analyze the following program fragment, where input x ranges over the integers {1,2,3,4,5}:

```
y = 3*x;
if y > 20 then ...
```

If we abstract each variable to an interval over the integers, then we can precisely represent x in the initial and final state of this simple program by the interval [1,5] which gives a minimum and maximum value that the variable can take on in any execution of the program. However, to reflect the action of multiplying by 3, the abstract domain would multiply the interval by 3, resulting in the interval [3,15] for y. Clearly, we have lost information in this abstraction, since we can only represent the possible values of y via a simple interval (rather than the precise set {3,6,9,12,15}). On the other hand, the abstraction does allow us to cheaply compute some kinds of information about the concrete program. In the example, we can symbolically evaluate the condition to false in the abstract domain, so we know that the then-branch can never be executed.

An abstract interpretation engine comes with a library of standard abstract domains that can be used to analyze a broad range of problems. The most common abstract domain is numeric intervals, as illustrated above, which abstract the possible values of a numeric variable at a program control point by a bounding interval. Another common abstract domain uses a set of linear constraints (i.e. an enclosing polyhedron) to over-approximate the joint values of several variables. The interval domain is computationally cheap and scales well to very large programs, although it can become imprecise. In contrast, a polyhedral abstract domain provides good precision since it relates several variables, but can be very expensive to apply.

An abstract domain is used to generate a specific kind of invariant at each program control point (e.g. the control point between assignments in C). The interval domain infers interval invariants for each program variable at each control point. The polyhedral domain infers a polyhedral envelope for some of the program variables at each control point.

To analyze a given program, an abstract interpreter starts with an abstract representation of the input and forward simulates that representation through the abstracted actions of the program. The simulation continues until a fixpoint is reached, which is expressed as a sound invariant at each control point in the program. The presence of loops complicates the analysis, since it may

be necessary to simulate around a loop an unlimited number of times. Abstract interpretation uses a widening operator to speed up convergence of the simulation to a fixpoint, by generalizing the current abstract representations. The result will generally be a fixpoint (set of invariants at control points) that is not a least fixpoint (the strongest expressible set of invariants at control points of the program).

Part of the art of abstract interpretation is carefully choosing where in a program to apply which abstract domain – where precision is needed (as in an inner loop) we might apply a polyhedral domain, but elsewhere use intervals. The goal is to gain as much information as possible about the program while minimizing analysis time and space. Another part of the art of abstract interpretation is developing new abstract domains that are tailored to a special class of programs. Often a new domain is motivated by a new kind of invariant that is need to effectively analyze and prove properties of a special class of program. In this project, with a special focus on linear digital filters and linear accumulators, it is natural to explore new abstract domains that generate appropriate invariants for those classes.

At the outset of this project, our working hypothesis was that the standard abstract domains used in abstract interpretation (including numeric intervals) would give results that are too imprecise to be useful. Consequently, the project goal was to explore abstract domains that are specialized to linear digital filters. As discussed later in Section 4, our working hypothesis was borne out, but, surprisingly, as we developed and applied more specialized domains, we discovered techniques that produce the desired analytic results without the need for abstract interpretation. We discovered techniques for generating closed-form formulas for LDFs and accumulator codes that yield sound and reasonably precise bounds on the output stream and its floating point errors. That is, we were able to obtain our desired analytic results in negligible time.

3.3. Integration of Model Analysis and Code Analysis

This section describes the original, standalone platforms of the two tools, and the modifications and bridges that we designed to allow their SANA-enhanced versions to work as one.

3.3.1. Original HiLiTE Platform

Figure 7 is an overview of the HiLiTE tool and its usage context, as deployed in the avionics certification programs in Honeywell aerospace products. HiLiTE provides comprehensive design model analysis and test generation for MATLAB Simulink/Stateflow models. It performs symbolic analyses combining computation semantics, control transforms, and temporal properties in a unified analysis framework. Error propagation is a recent addition to support increasing tolerances only where necessary and identify when a construct is untestable because cumulative error makes an execution path undecidable.

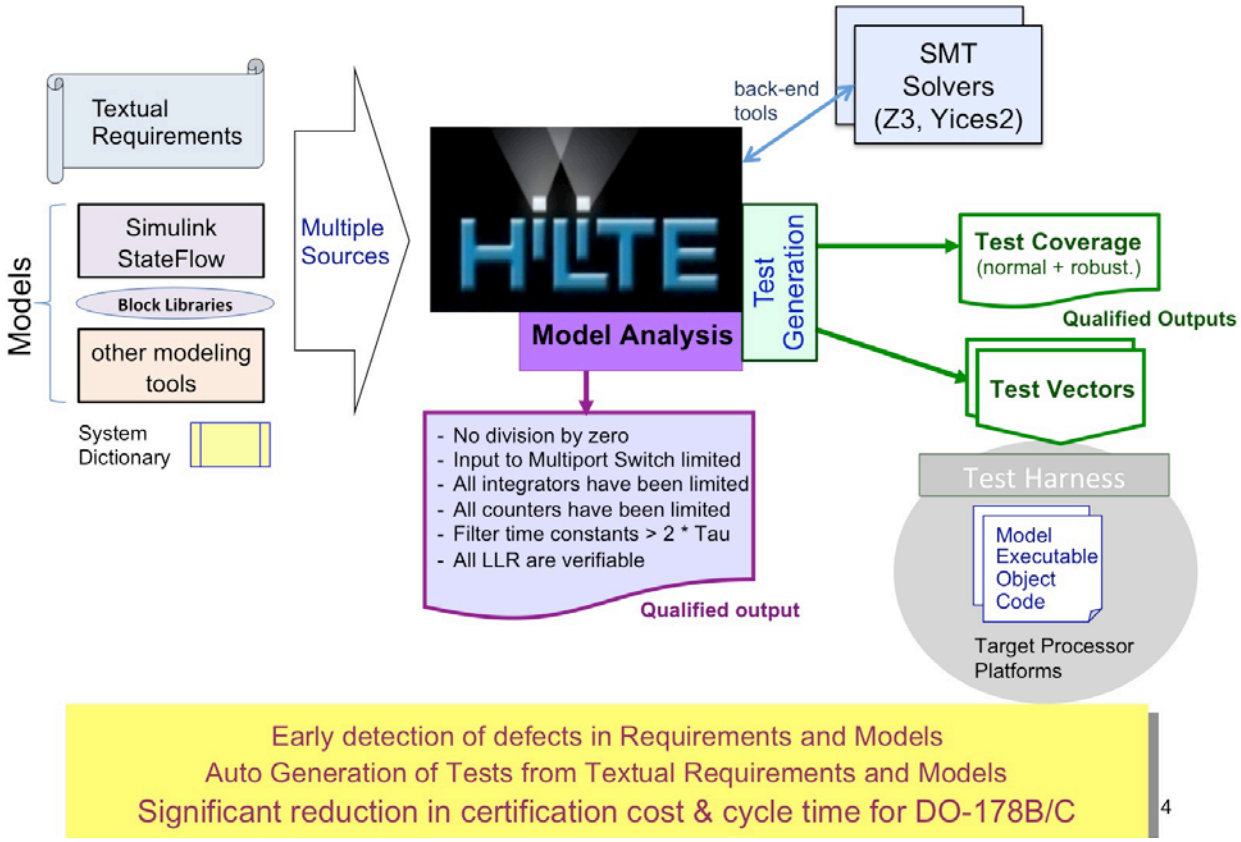

Figure 7. HiLiTE Overview

Range propagation: During model analysis, HiLiTE propagates signal dimensions, data type, and range information from the model input blocks through the rest of the blocks in the model to the model outputs. In this propagation, the dimensions, data type, operating range, and hard range constraint are determined for all intermediate signals (all inputs and outputs of each block instance) in the model. The range computation takes into account the specific mathematical and functional effect of each library block. The following are computed for each output of each block:

- Shape: The dimensions of the signal represented as a list of sizes for each dimension; scalar signals have shape=1.
- Data type: The data type is inferred from the block type and input data types.
- Normal operating range: The range of possible values the output can have, for all combinations of input values of the block within the normal operating range of each input. This may include an estimated error component on each end of the range.
- Maximum allowable range constraint (also called a “hard bound” or “constraint”): This denotes a constraint on the feasible values due to specific block computations. Examples are outputs of the constant and range limiter blocks; values on these signals outside the constraint are not feasible in the model.

Based on the range analysis certain model design defects can be analyzed including overflow conditions, frozen signals (constant value), un-testable conditions, and violation of modeling design guidelines or block-specific constraints.

Feedback loops and patterns: HiLiTE detects and breaks any feedback loops and performs the propagations (Shape/DataType/Range) for all the blocks interconnected in the loop. Similarly, it analyzes certain combinations of blocks to identify common patterns. HiLiTE substitutes a single block for the model blocks that comprise the pattern and performs the comprehensive propagation analysis and test vector generation for the entire pattern as a whole.

Relationship propagation: HiLiTE has the capability to determine the relationship of a block's output to the upstream source ports in the model. A constraint in the model occurs when a signal fans out, creating multiple branches that converge downstream onto the inputs of a single block. When two or more such inputs are totally related to each other (identity or linear relationship with zero slope) HiLiTE recognizes this and maintains a list of totally related inputs for the blocks in the model. If the output of the block is a polynomial of the block inputs HiLiTE recognizes the polynomial relationship and uses it to determine a tight range bound at the output.

The relationship propagation takes into account the specific mathematical and functional effect of each library block. The propagation of these relationships supports the analysis of the data flow expression to recognize and solve the equation to derive tight, exact range bounds at a block's output. Also it helps in identifying test generation constraints due to related inputs in the model.

Feedback loop invariant analysis: We extended HiLiTE's feedback loop and relationship processing to discover cycle invariants describing cycle input-state-output behaviors. A cycle invariant is expressed by guard/assignment pairs, where each pair represents a possible execution path of the cycle. HiLiTE discovers cycle invariants by performing path analysis and integration of individual block invariants following the analysis procedures described below.

Figure 8 shows a model of a limited counter with only one cycle. This cycle has three possible paths due to the three different guard conditions of block *limit*. For each path, we collect the set of guard/assignment pairs from blocks within the cycle starting with the time-dependent block *delay*, followed by *sum* and *limit*.

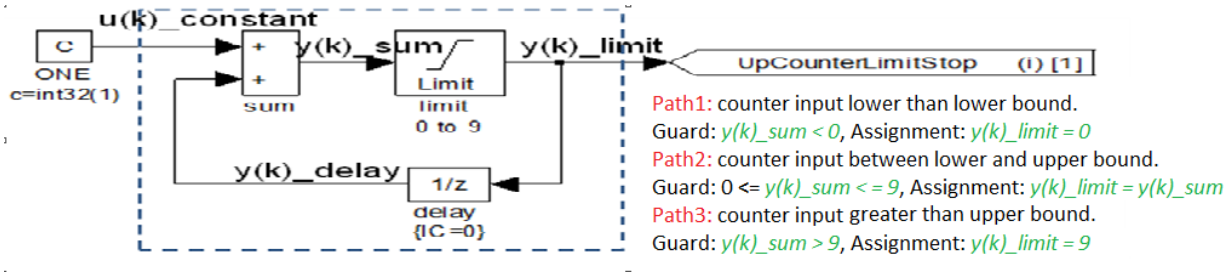

Figure 8. Simulink model of limited counter and three possible paths

Next, we use SMT solver Z3 to check the satisfiability of each path. HiLiTE automatically generates the input file for Z3 containing variable declarations, sets of constraints translated from the block level guard/assignment pairs and some checking commands. Z3 returns "unsat" for

each invalid path, which is then removed from the cycle invariant. For the remaining valid paths (all 3 paths are valid in Figure 8), we further reduce the block level guard/assignment pairs into a cycle level guard/assignment pair possessing only the cycle level input/output variables. This process includes virtual substitution to eliminate the intermediate variables as well as some general simplification. Finally, the cycle invariant is printed out in the log.

The OptionSet information to include in the command file looks like this:

```
<OptionSet name="ModelAnalysisDirectives">
     <Option key=" AnalyzeFeedbackLoops">True</Option>
 </OptionSet>
```

By running HiLiTE on a model with the option "AnalyzeFeedbackLoops" set as “True,” HiLiTE obtains the cycle invariant and decoupler outport range. In the analysis process, messages are given as information including “cycle list information,” “path expressions,” and “range calculated for the time-dependent block outport in current cycle list”.

3.3.2. Original CodeHawk Platform

Kestrel Technology’s CodeHawk technology is an Abstract Interpretation framework, written in Ocaml, that is meant to be specialized to particular programming languages and program properties of interest. A specific analyzer tool is built for a specific combination of language/properties by specializing the Ocaml source and compiling to the intended operating system (Figure 9 below). Thus CodeHawk can in principle run on any machine/OS for which there is an Ocaml compiler. We have standard front ends for C, Java and x86 binaries, and a variety of abstract domains over which to approximate program behaviors. For this project, we started with an exiting specialization for memory safety properties of C programs that typically runs on Unix/Linux/MacOS. The need to extend this analyzer to a new floating-point abstract domain during the course of this project caused us to re-implement a portion of the analyzer in C++, thus breaking the monolithic Ocaml compilation and resulting in a mixed language system that (currently) runs only on MacOS (see 3.3.4 below).

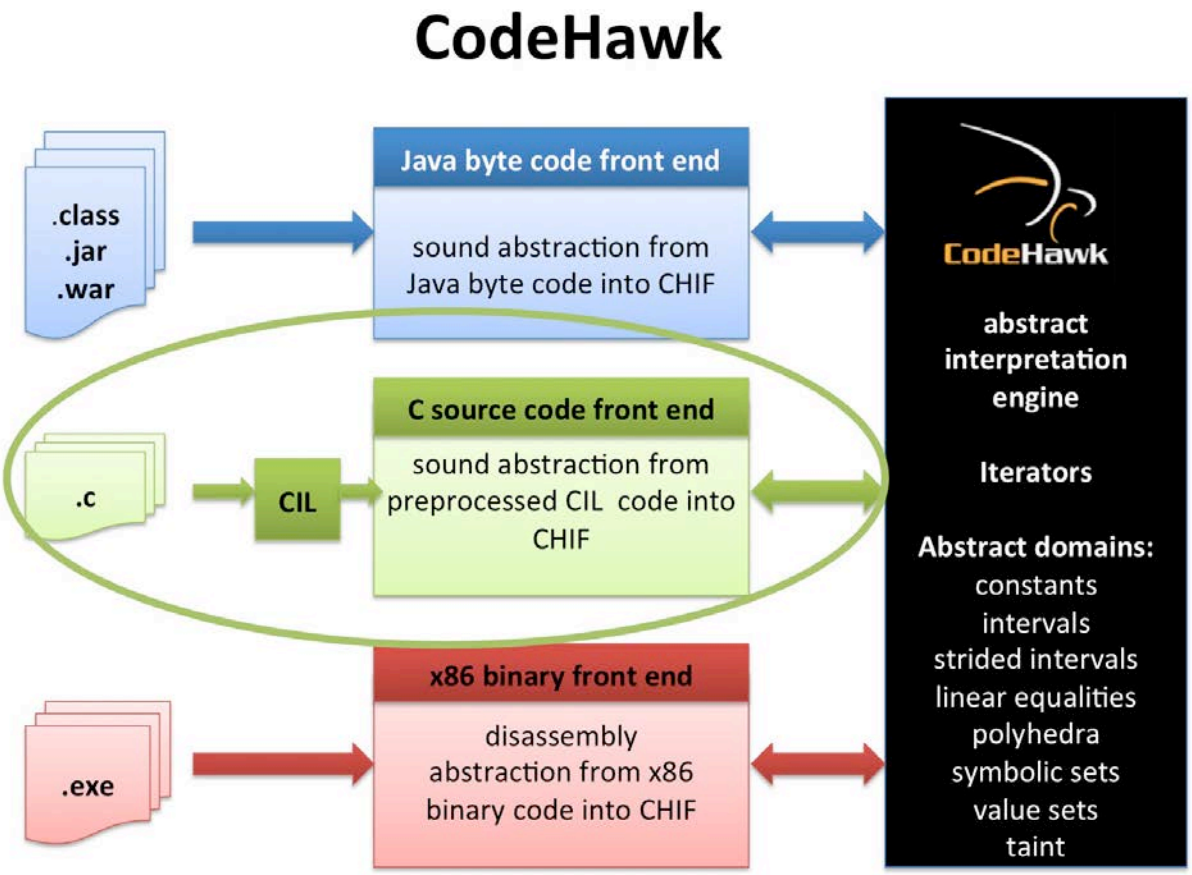

Figure 9 CodeHawk Overview

3.3.3. JSON Interchange Architecture

Since HiLiTE and CodeHawk typically run on different operating system platforms, we made an early architectural decision to exchange information between their SANA-enhanced versions via an external file interface. This would allow development to proceed in parallel at different sites, and for the final, integrated deliverable to be run on the same or different machines at the same site, as well as different machines at different sites. We designed a custom JSON interface specification that was sufficiently abstract to mask differences between the two tools and their respective source languages. As the first year enhancements and integration progressed, we repeatedly revised this common JSON specification to match, buffering the changes with grammars and parser generators to rebuild the necessary translators.

Figure 10 shows the overall file exchange integration for Linear Digital Filters. CodeHawk normally takes as input all of the C sources required to compile the filter application (or any other C application). For this project, we restricted the C source to just that implementing the actual filter function that HiLiTE generates. In addition to the C sources, HiLiTE also passes model-level information about the filter (parameters, recurrence relation, input bounds, etc) in a JSON file, which contains a reference to the C source file. CodeHawk consumes both inputs, uses the model-level JSON information to specialize the abstract interpretation to the input filter,

then writes back an approximation of the value and error bounds of the filter's output to the same JSON instance. HiLiTE then consumes this rewritten instance to exploit the computed bounds at the filter model level.

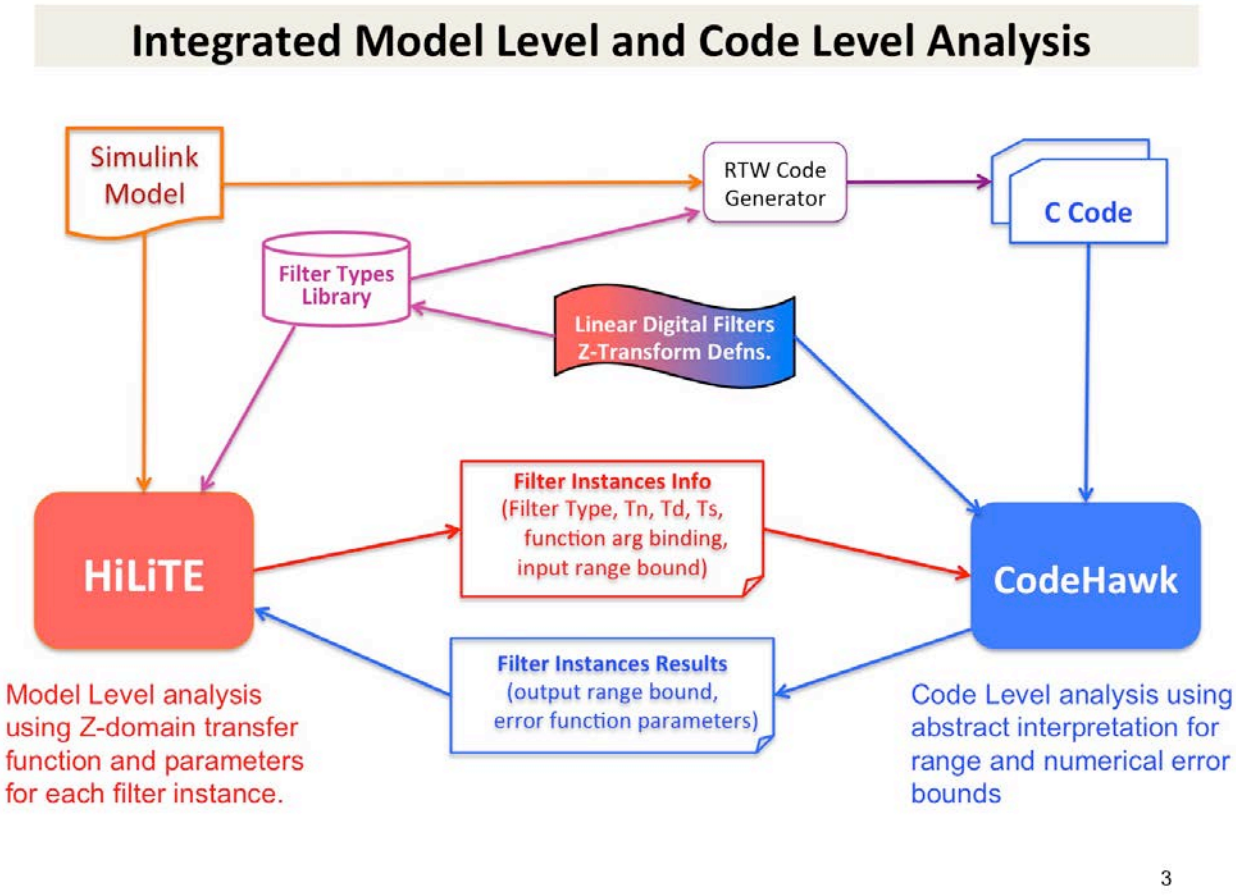

Figure 10 HiLiTE <-> CodeHawk Integration via files

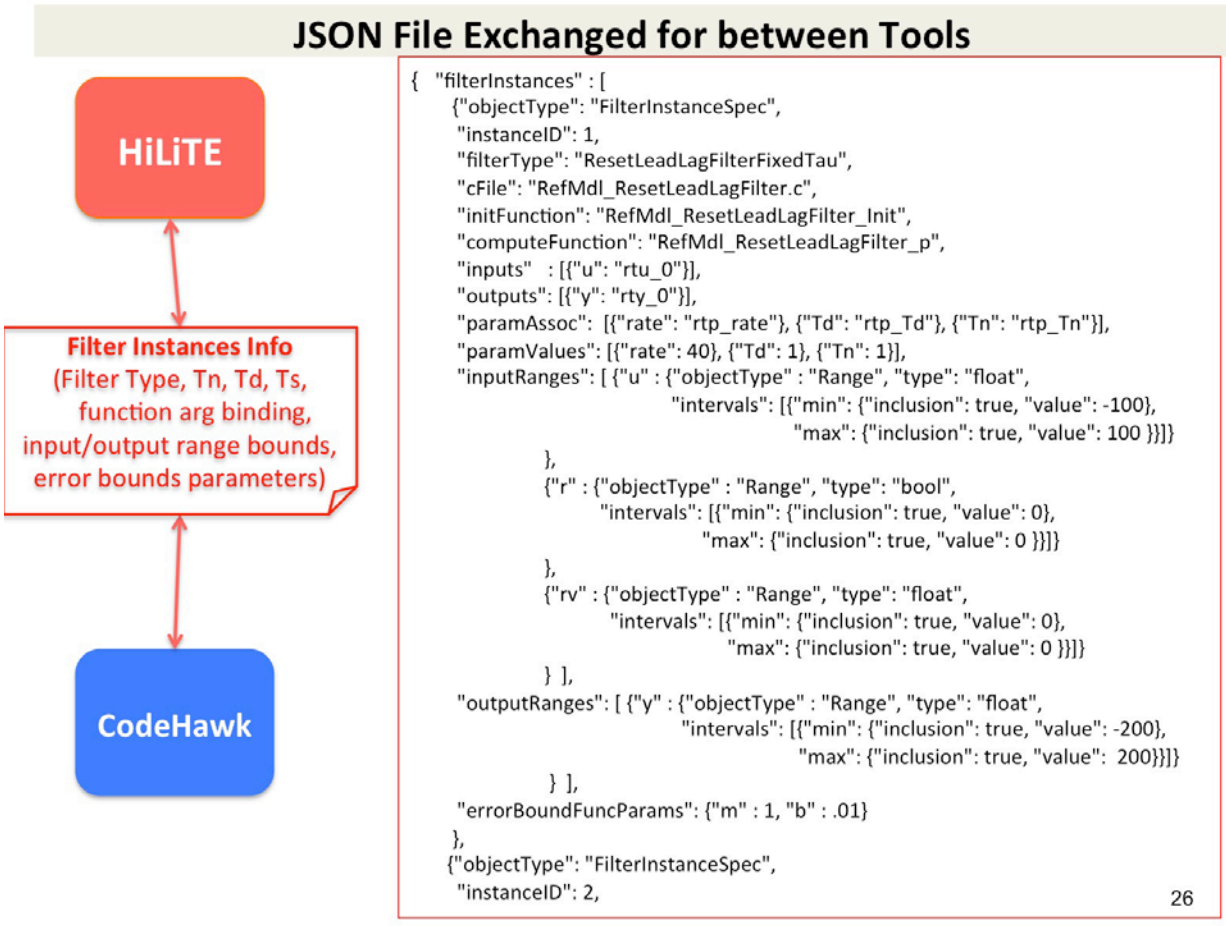

Figure 11 HiLiTE <-> CodeHawk JSON Exchange Example

Figure 11 illustrates a sample JSON input file (portion thereof) for a filter analysis. This file contains information about the filters found in a model. A given C application may contain a number of filter functions, and any one filter function may be used by multiple "instances" of that filter. The JSON file for filter analysis consequently uses a series (JSON array) of filterInstanceSpec objects. Each instance represents a particular set of parameters for the filter, theoretical bounds on the inputs to the filter, references to the C sources and their algorithmic parts (init and step functions) that implement the filter, and placeholder JSON objects representing the value and error bounds on the outputs from the filter. CodeHawk will consume the input characteristics, and rewrite the JSON output objects to record the results of analysis.

Note that the HiLiTE JSON output refers to other files. These files may be the source code for standard functionality referenced by the analyzed model or it may be the source code generated for the model itself.

The JSON format for integrating accumulators explored in the second year of this project is different than that for filters since different information is required for these two types of analyses. We avoid confusion in the exchange both by using different files for each type of analysis data and by the creation of a 'filterInstances' list for filter analysis data and an 'accumulatorInstances' list for accumulator analysis data.

3.3.4. Tool Software Integration Architecture

The toolset integration has several ‘levels’. There is the integration of the entire system of tools (CodeHawk + HiLiTE) as well as the integration of each of the tool’s internal systems within itself.

HiLiTE is a Windows .NET application comprised of multiple C# and J# assemblies, some statically linked and others dynamically loaded. In addition, HiLiTE links to COM libraries for its interface to MATLAB Simulink to open and process the contents of a model.

CodeHawk is normally an OCaml application specialized to a particular source language (C, Java, x86 binary) and a particular set of program conjectures to be proved. This project required specializing Kestrel’s existing C analyzer to a new abstract domain of floating point numbers. The libraries for this domain, however, are written in C++, so our original architectural plan was to write Ocaml “wrapper” code for these C++ libraries to bring them into the existing Ocaml architecture. This proved to be infeasible, so we were forced to write an entirely new abstract interpretation engine in C++ and split the erstwhile monolithic CodeHawk architecture into several pipelined pieces. Because of the exigencies of developing and integrating these pieces from multiple languages at multiple sites, the resulting SANA-specialized CodeHawk (currently) only runs reliably under MacOS. Its constituent pieces are assembled from C, C++, and Ocaml compiled binaries, plus a Unix shell script, and integrated into a single application as a runnable Java jar file.

A consequence of this mixed-language/mixed-operating-system configuration is that the combined HiLiTE/CodeHawk system typically runs on separate Windows and Mac machines, with round-trip communication implemented via shared C source and JSON files.

The source code for the library of standard functionality that can be referenced is installed as part of the CodeHawk installer. Currently this set of standard functionality consists of all the known filters.

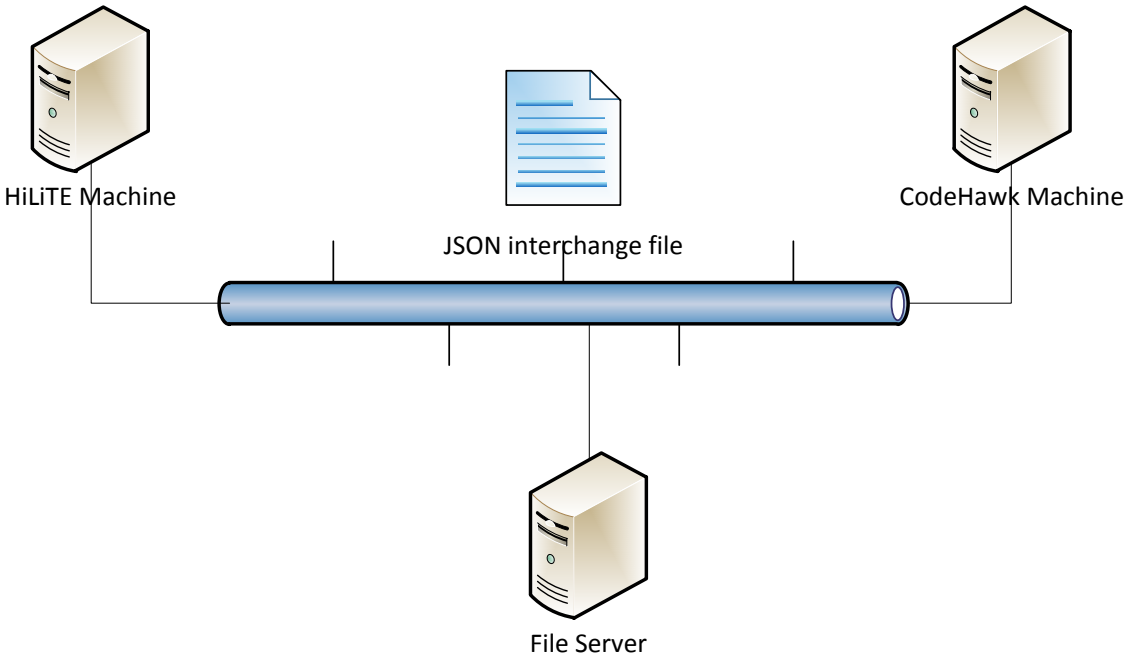

Figure 12 HiLiTE <-> CodeHawk Comm Process

The system of tools is integrated through the use of the files described above. Since these JSON files also reference 'C' source files, those files must either already be present on the CodeHawk machine or must be transferred along with the JSON file that references them. The filter source code is installed along with CodeHawk and thus need not be transferred. HiLiTE recognizes this and bundles model specific source code along with the JSON in a package that can be easily made accessible to the CodeHawk machine.

Because the HiLiTE machine already has all the source code, the only information that need be transmitted back from the CodeHawk machine is the result JSONfile.

The actual mechanism for transfer can be through use of a file server, email or any other mechanism convenient to the user. As part of moving files, they will need to be placed where CodeHawk/ HiLiTE can find them. See the User Guide for details.

4. RESULTS AND DISCUSSION

4.1. Discoveries in Abstract Interpretation

As discussed in the previous section, our approach was to explore various abstract domains within an abstract interpretation framework, implemented by CodeHawk, for analyzing linear digital filters and accumulating integrators. The analytic results that we desire are

(1) bounds on the output of a filter,
(2) bounds on the floating-point errors in the output of a filter,
(3) bounds on the output of an accumulating integrator as a function of the number of iterations, and
(4) bounds on the floating-point errors in the output of an accumulating integrator as a function of the number of iterations.

4.1.1. Interval Abstract Domain

As always in analyzing a program, the first question is what kinds of invariants are required to establish the desired results. Since we obviously want bounds as results, the first abstract domain to try is intervals.

As a running example, we will use a simple first-order linear digital filter (called a Lead Lag filter, see Section 4.2.1):

$$y_{n+1} = \frac{9}{13}u_{n+1} - \frac{7}{13}u_n + \frac{11}{13}y_n \tag{3}$$

where the input stream is written $u_0, u_1, \ldots$ and the output stream is $y_0, y_1, \ldots$ We are given that the range of the input values is [-1,1], and that $y_0 = 0$. What are the possible values for the output stream? If we use the interval abstract domain, representing y_0 as [0,0] and iterate the recurrence we get

$$y_1 = \frac{9}{13}[-1,1] - \frac{7}{13}[-1,1] + \frac{11}{13}[0,0] = [\frac{-16}{13}, \frac{16}{13}] \tag{4}$$

$$y_2 = \frac{9}{13}[-1,1] - \frac{7}{13}[-1,1] + \frac{11}{13}\left[\frac{-16}{13}, \frac{16}{13}\right] = \left[\frac{-192}{169}, \frac{192}{169}\right] \tag{5}$$

Note that the sequence is growing and we have gone through the iterative loop twice. A typical step in abstract interpretation is to speed up convergence to a fixpoint by applying a widening operator. There are standard widening operators for each abstract domain and their choice is somewhat of an art. A widening operator suggested in earlier work on filters [3] is to jump up to the nearest power of ten, so [-10,10] in our case. This does in fact converge since

$$\frac{9}{13}[-1,1] - \frac{7}{13}[-1,1] + \frac{11}{13}[-10,10] \subseteq [-10,10]. \tag{6}$$

The actual range of the output is [-1,1], so this inferred range is very imprecise, although sound. The best that can be obtained using the interval domain is [-8,8] which is better, but far from the exact bound.

4.1.2. Cancellation Abstract Domain

The essential reason for the imprecision when using the interval domain is that there are cancellation possibilities if we simply unroll the recurrence:

$$\begin{aligned} y_2 &= \frac{9}{13}u_2 - \frac{7}{13}u_1 + \frac{11}{13}y_1 \\ &= \frac{9}{13}u_2 - \frac{7}{13}u_1 + \frac{11}{13}\left[\frac{9}{13}u_1 - \frac{7}{13}u_0 + \frac{11}{13}y_0\right] \\ &= \frac{9}{13}u_2 + \frac{8}{169}u_1 - \frac{77}{169}u_0 \end{aligned} \tag{7}$$

where some of the contribution of u_1 to y_2 has been cancelled out. This phenomenon suggests an approach to constructing an abstract domain that generates stronger invariants for linear recurrences: supplement intervals with symbolic expressions to allow cancellation as part of the abstract operators of the domain. As an example, instead of abstracting y_2 to an interval, we abstract it to a symbolic sum (where *Abs(x)* denotes the abstract value of variable *x*):

$$Abs(y_2) = \frac{9}{13}Abs(u_2) + [-\frac{8}{169}, \frac{8}{169}] \tag{8}$$

which will allow us to partially cancel $Abs(u_2)$ in the next iteration (to generate $Abs(y_3)$). We implemented an abstract domain of this type and indeed it converges to the exact bound [-1,1] in our example. It generally performs well for first-order linear filters since there are a fixed number of terms to be carried between iterations to harvest all the cancellation possibilities. Unfortunately, for higher-order linear filters, there is no fixed bound on the number of terms to be carried so this approach won't work as stated. We explored several ways to further elaborate the approach for second-order filters, but the result is very complex and of unclear benefit.

Fortunately, the idea of unrolling the recurrence carries the seed of an alternate approach that proved successful. Intuitively, we can imagine fully unrolling the recurrence until any output y_n is expressed as a linear sum of all preceding inputs (i.e. all previous outputs in the recurrence are eliminated by unrolling). Before we turn to the approach based on that concept, we first review our parallel efforts to analyze the error due to floating-point representations.

4.1.3. Analyzing for Error Bounds

In addition to wanting bounds on the values of the output of a filter or accumulator, we also want to know bounds on the accumulated errors due to floating point implementation of real values. Using simple intervals for error analysis will result in a non-converging series of bounds. In practice the errors are bounded and fairly small, at least for first-order filters, so once again the interval domain proves to be too imprecise for this application.

As mentioned earlier, if the standard abstract domains do not provide a useful time/precision tradeoff, then the skilled abstract interpretationist explores the type of invariants that are required to obtain the desired results and develops abstract domains to generate those invariants.

We define the error as

$$f(x,n) - r(x,n) \leq \epsilon(x,n) \tag{9}$$

where $r(x,n)$ denotes the real value of variable *x* at the *n*th iteration, $f(x,n)$ is the floating point value of *x* after *n* iterations, and $\epsilon(x,n)$ is the error, expressed as a bound on the difference of the two.

If we view the output y_n as a linear sum of all preceding inputs, we get a sense that the net error due to floating point can be calculated as a linearly weighted sum of the error contributions of each of the inputs and the operations performed upon them. This leads, after some trial and error, to seeking an invariant of the form

$$\epsilon(x,n) = A_x + M_x b^{p_x} \sum_{i=0}^{n} \beta^i \tag{10}$$

where $\epsilon(x,n)$ is the error range of variable *x* at a code location. The sum conveys the effect of the errors from all earlier inputs. The invariant is parametric on *A, M, p, b,* and β, where A_x, M_x, and p_x vary per variable *x*. As we simulate the concrete program (implementing the recurrence) via abstract operators, the effect will be to update the current abstract values of A_x, M_x, and p_x .

We have been able to derive the abstract operations that preserve the form of the invariant by changing the parameters. To illustrate, consider the case that the concrete program performs a sum of two variables

$$v = w + z \tag{11}$$

where the invariant holds before the assignment. We want to calculate updates to the parameters of the invariant such that the invariant holds after the assignment.

We assume that

$$\epsilon(w) = A_w + M_w b^{p_w} \sum_{i=0}^{n} \beta_w^i \tag{12}$$

$$\epsilon(z) = A_z + M_z b^{p_z} \sum_{i=0}^{n} \beta_z^i \tag{13}$$

then (letting ϵ_m denote the machine error for the chosen floating-point representation)

$$
\begin{aligned}
& f(v,n) - r(v,n) \\
=\ & \big(f(w,n) + f(z,n)\big)(1+\delta) - \big(r(w,n) + r(z,n)\big) \text{ where } |\delta| \le \epsilon_m \\
=\ & \big(r(w,n) + \epsilon(w) + r(z,n) + \epsilon(z)\big)(1+\delta) - \big(r(w,n) + r(z,n)\big) \\
=\ & \big(r(w,n) + r(z,n)\big)\delta + \big(\epsilon(w) + \epsilon(z)\big)(1+\delta) \\
=\ & r(v,n)\delta + \big(\epsilon(w) + \epsilon(z)\big)(1+\delta) \qquad (14) \\
\le\ & \max\big(r(v,n)\big)\,\epsilon_m + (A_w + M_w b^{p_w} \textstyle\sum_{i=0}^{n} \beta_w^i + A_z + M_z b^{p_z} \sum_{i=0}^{n} \beta_z^i)(1+\delta) \\
\le\ & [\max\big(r(v,n)\big)\,\epsilon_m + (A_w + A_z)(1+\epsilon_m)] \\
& + \frac{(M_w + M_z)(1+\epsilon_m)}{b}\, b^{\max(p_w,p_z)+1} \sum_{i=0}^{n} \beta_z^i \\
=\ & \epsilon(v,n)
\end{aligned}
$$

where we update the parameters according to

$$A_v = [\max\big(r(v,n)\big)\,\epsilon_m + (A_w + A_z)(1+\epsilon_m)] \qquad (15)$$

$$M_v = \frac{(M_w + M_z)(1+\epsilon_m)}{b} \qquad (16)$$

$$p_v = \max(p_w, p_z) + 1. \qquad (17)$$

That is, when the concrete implementation assigns a sum of two variables to a variable, we perform the updates above in the abstract domain. Similar abstract operators have been derived and implemented for other concrete operations: subtraction, multiply by a constant, divide, and so on.

Using this abstract domain resulted in finite error bounds that were surprisingly small. One might intuit that floating-point errors should accumulate without bound as a linear recursion progresses. However, it seems that filters inherently have a discounting mechanism so that the contribution of previous inputs increasingly diminish with age. The same mechanism seems to apply to errors: the effects of older errors are discounted with age, so that the age-weighted errors form a series that converges.

4.1.4. Closed-form Solutions

The explorations discussed in previous subsections pointed to the conceptual value of fully unrolling the recursion and expressing the current output y_n as a linear sum of all preceding inputs. Anca Browne discovered that the coefficients of that linear sum are fixed by the linear recurrence, and, moreover, are themselves characterized by a homogeneous linear recurrence relation. This remarkable fact allows us to find a closed-form expression of the coefficients and then to symbolically calculate exact bounds on the output values. A generalization of this approach can also be applied to calculating bounds on error values and to find bounding

functions on the range and errors of accumulating integrators. The effect is that through symbolic analysis of the linear recurrences that characterize linear digital filters and accumulating integrators, we can derive exact or tight bounds on the range of outputs and their floating point errors, in essentially constant time. This stands in contrast, say, to 20 hours of analysis time needed to perform sophisticated abstract interpretation-based analysis for just one Airbus-related analysis in [3].

Let

$$y_{n+1} = a_0 u_{n+1} + a_1 u_n + \ldots + a_N u_{n-N+1} + b_1 y_n + \ldots + b_N y_{n-N+1} \quad (18)$$

be the recurrence equation for the output of a digital filter of order N that defines the output y_{n+1} in terms of N + 1 present and past inputs and N past outputs. Unfolding the equation, y_{n+1}can be expressed as a linear combination of u_{n+1} ... u_1:

$$y_{n+1} = c_0^{n+1} u_{n+1} + c_1^{n+1} u_n + \cdots + c_n^{n+1} u_1 \quad (19)$$

Notice that for any *n*, $c_0^{n+1} = a_0$. Therefore for any n, $c_1^{n+1} = a_1 + b_1 c_0^n = a_1 + b_1 a_0$, etc. The key point to notice is that c_k^n does not depend on *n*, for any *k*. Without the superscript,

$$y_{n+1} = c_0 u_{n+1} + c_1 u_n + \ldots + c_n u_1 \quad (20)$$

The first N+1 terms $c_0, \ldots, c_N$ depend on both $a_0, \ldots, a_N$ and $b_1, \ldots, b_N$. However, for any $n \geq N$,

$$c_{n+1} = b_1 c_n + b_2 c_{n-1} + \ldots + b_N c_{n-N+1} \quad (21)$$

This homogeneous linear recursion formula of order N has the general solution

$$c_{n+1} = P_1(n) r_1^n + P_2(n) r_2^n + \ldots + P_j(n) r_j^n \quad (22)$$

where $r_1, \ldots, r_j$ are solutions of the characteristic equation

$$x^N = b_1 x^{N-1} + b_2 x^{N-2} + \ldots + b_N \quad (23)$$

with respective multiplicities $m_1, m_2, \ldots, m_j$ and P_i's are polynomials of degree $m_i - 1$. See e.g. [4] for more background on solving (in)homogeneous linear recurrence equations.

The initial conditions that determine the solution are given by $c_1, \ldots, c_N$.

4.1.5. Range Analysis for First-order Linear Digital Filters

The general form of a first-order linear filter is

$$y_0 = 0 \tag{24}$$

$$y_{n+1} = a_0 u_{n+1} + a_1 u_n + b_1 y_n \text{ for } n \geq 0 \tag{25}$$

where $u_0 = 0$. The coefficients c_i are given by

$$c_0 = a_0 \tag{26}$$

$$c_1 = a_1 + b_1 a_0 \tag{27}$$

$$c_{n+1} = c_1 b_1^n \tag{28}$$

and therefore the output is

$$y_{n+1} = a_0 u_{n+1} + (a_1 + b_1 a_0)(u_n + b_1 u_{n-1} + \cdots + b_1^{n-1} u_1) \tag{29}$$

or

$$y_{n+1} = a_0 u_{n+1} + (a_1 + b_1 a_0) \sum_{i=0}^{n-1} b_1^i u_{n-i} \tag{30}$$

Since we are assuming that all inputs have the same range, we can omit the subscript on inputs for purposes of computing the range bounds of the output:

$$\begin{aligned} max(y_{n+1}) &= max(a_0 u) + max((a_1 + b_1 a_0) \sum_{i=0}^{n-1} b_1^i u) \\ &= max(u)(a_0 + \left((a_1 + b_1 a_0) \cdot \tfrac{1 - b_1^n}{1 - b_1}\right)) \\ &\leq max(u)\left(a_0 + \left((a_1 + b_1 a_0) \cdot \frac{1}{1 - b_1}\right)\right) \end{aligned} \tag{31}$$

when $b_1 < 1$. Similarly,

$$min(y_{n+1}) \leq min(u)\left(a_0 + \left((a_1 + b_1 a_0) \cdot \frac{1}{1 - b_1}\right)\right).$$

As an example, consider the lead lag filter (see Section 4.2.1) described by

$$y_0 = 0 \tag{32}$$

$$y_{n+1} = \frac{9}{13} u_{n+1} - \frac{7}{13} u_n + \frac{11}{13} y_n \tag{33}$$

with $u_0 = 0$ and for any $i > 0,\ u_i \in [-1, 1]$. Then the coefficients c_i are given by

$$c_0 = \frac{9}{13} \tag{34}$$

$$c_1 = \frac{7}{13} + \frac{11}{13}\frac{9}{13} = \frac{8}{169} \tag{35}$$

$$c_{n+1} = \frac{8}{169}\left(\frac{11}{13}\right)^n \tag{36}$$

and the range of the output by

$$max(y_n) \leq \frac{9}{13} + \frac{8}{169}\frac{1}{1-\frac{11}{13}} = 1 \tag{37}$$

$$min(y_n) \leq -\frac{9}{13} - \frac{8}{169}\frac{1}{1-\frac{11}{13}} = -1 \tag{38}$$

4.1.6. Range Analysis for Second-order Linear Digital Filters

A similar, slightly more complex treatment computes the range bound for second-order linear filters:

$$y_0 = y_1 = 0 \tag{39}$$

$$y_{n+1} = a_0 u_{n+1} + a_1 u_n + a_2 u_{n-1} + b_1 y_n + b_2 y_{n-1}\ for\ n \geq 1. \tag{40}$$

Suppose that the characteristic equation has two distinct solutions r_1 and r_2. Then

$$c_0 = a_0 \tag{41}$$

$$c_1 = a_1 + b_1 a_0 \tag{42}$$

$$c_2 = b_1 + b_1^2 a_0 + b_2 a_0 \tag{43}$$

$$c_{n+1} = s r_1^n + t r_2^n \tag{44}$$

where *s* and *t* are the solutions of

$$c_1 = s + t \tag{45}$$

$$c_2 = s\, r_1 + t\, r_2 \tag{46}$$

The output is

$$y_{n+1} = a_0\, u_{n+1} + (s + t)\, u_n + (s\, r_1 + t\, r_2)\, u_{n-1} + \ldots \tag{47}$$

In the case when s, t, r_1, r_2 are all positive the range is given by

$$max(y_n) = max(a_0\, u) + s(max(u))(1 + r_1 + r_1^2 + \ldots) + t(max(u))(1 + r_2 + r_2^2 + \ldots) \tag{48}$$

$$min(y_n) = min(a_0\, u) + s(min(u))(1 + r_1 + r_1^2 + \ldots) + t(min(u))(1 + r_2 + r_2^2 + \ldots) \tag{49}$$

For example, consider the following quadratic filter (discussed in Section 4.2.4)

$$y_0 = y_1 = 0 \tag{50}$$

$$y_{n+1} = \frac{592841}{78010601} u_{n+1} + \frac{1185682}{78010601} u_n + \frac{592841}{78010601} u_{n-1} + \frac{126814318}{78010601} y_n - \frac{51175081}{78010601} y_{n-1} \tag{51}$$

where $u_0 = 0$ and for any $i > 0,\ u_i \in [-100, 100]$.

The first coefficients are

$$c0 = \frac{592841}{78010601} \tag{52}$$

$$c1 = \frac{167676492512320}{6085653868381201} \tag{53}$$

$$c2 = \frac{225048660239351545020 80}{474745515750392387111801} \tag{54}$$

and the others are determined by the solutions of the characteristic equation

$$x^2 = \frac{126814318}{78010601} x - \frac{51175081}{78010601} \tag{55}$$

which are

$$r_{1,2} = \frac{63407159 \pm 160\sqrt{1104257321}}{78010601} \approx \{0.744646, 0.880957\} \tag{56}$$

The general solution is

$$c_{n+1} = s\left(\frac{63407159 - 160\sqrt{1104257321}}{78010601}\right)^n + t\left(\frac{63407159 + 160\sqrt{11042573211}}{7801060}\right)^n \tag{57}$$

where s and *t* are solutions of the system

$$s + t = c_1 \tag{58}$$

$$s\left(\frac{63407159 - 160\sqrt{1104257321}}{78010601}\right) + t\left(\frac{63407159 + 160\sqrt{1104257321}}{78010601}\right) = c_2 \tag{59}$$

which are

$$s = -\frac{5579860(-976009979996381 + 391157262321\sqrt{1104257321})}{3953016374842300954719 1}} \approx -0.169695 \tag{60}$$

$$t = \frac{5579860\sqrt{\frac{17}{64956313}}(391157262321 + 883861\sqrt{1104257321})}{6085653868381201} \approx 0.197247 \tag{61}$$

The expansion of y_{n+1} is

$$y_{n+1} = c_0 u_{n+1} + (s+t)\,u_n + (s\,r_1 + t\,r_2)u_{n-1} + \cdots + (s\,r_1^{n-1} + t\,r_2^{n-1})u_1 \tag{62}$$

Notice that $s\,r_1^k + t\,r_2^k > (s+t)r_2^k > 0$ since $s < 0$, $s + t > 0$ and $r_2 > r_1$. As all the coefficients are positive, the maximum is attained for $max(u)$ and the minimum for $min(u)$

$$\begin{aligned} max\,(y_n) &= 100\,(c_0 + (s+t) + (s\,r_1 + t\,r_2) + \ldots) \\ &= 100\,(c_0 + s\,\frac{1}{1 - r_1} + t\frac{1}{1 - r_2}) = 100 \end{aligned} \tag{63}$$

$$\begin{aligned} min\,(y_n) &= -100\,(c_0 + (s+t) + (s\,r_1 + t\,r_2) + \ldots) \\ &= -100\left(c_0 + s\,\frac{1}{1 - r_1} + t\frac{1}{1 - r_2}\right) = -100. \end{aligned} \tag{64}$$

4.1.7. Digital Filters: Error Range

The filter recursive formula does not immediately translate into a constant-term recursive formula for the floating-point output or its accumulated error. Different inputs, with different errors, will influence the constants differently. The solution we adopt is to use intervals. However, just reproducing our approach for finding the real bounds does not work because the characteristic equation cannot be solved in the same way over intervals.

As before, the error for variable x is

$$f(x,n) - r(x,n) \leq \epsilon(x,n) \tag{65}$$

where $r(x,n)$ denotes the real value of variable *x* at the *n*th iteration for some particular sequence of inputs, $f(x,n)$ is the floating point value of *x* after *n* iterations, and $\epsilon(x,n)$ is the error, expressed as a bound on the difference of the two. We will assume that all the variables are bounded over all iterations and we will denote by $R(x)$ an interval that includes all possible real values for x and by $EV(x)$ an interval that includes all the possible errors for variable x. For inputs u_i, we will take $EV(u_i)$ to be a symmetric interval.

4.1.7.1. Symbolic Interval Combinations

Let $\mathcal{I}$ be the set of intervals over real numbers. Let $\mathcal{S}$ be a set of symbols and $\mathcal{A}$ be the set of affine combinations

$$G_1 S_1 + \ldots G_p S_p + G \tag{66}$$

where $G_0, \ldots G_p, G \in \mathcal{I}$ and $S_0, \ldots, S_p \in \mathcal{S}$. Two combinations that differ only by the order of their terms or by a zero-coefficient term as in

$$G_1 S_1 + \ldots G_p S_p + G \equiv G_1 S_1 + \ldots G_p S_p + [0,0] S_{p+1} + G \tag{67}$$

are considered identical.

We will assume that $\mathcal{S}$ contains all the symbols that we need, among which are symbols for the infinite sequence of input errors and output errors.

We define the following operations on $\mathcal{A}$:

1. multiplication by an interval

$$(G_1 S_1 + \ldots G_p S_p + G) * I = (G_1 * I) S_1 + \ldots (G_p * I) S_p + G * I \tag{68}$$

2. division by an interval

$$G_1 S_1 + \ldots G_p S_p + G)/I = (G_1/I) S_1 + \ldots (G_p/I) S_p + G/I \tag{69}$$

3. addition

$$\begin{aligned}(G_1\,S_1\ +\ \ldots\ G_p\,S_p\ +\ G)\ +\ \left(G'_1\,S_1+\ldots G'_p\,S_p\ +\ G'\right)\\ =\ (G_1+G'_1)\,S_1+\ldots(G_p+G'_p)\ S_p\ +(G+G')\end{aligned}\tag{70}$$

Notice that we can always assume two combinations have the same symbols because we can always add zero-coefficient terms.

4. subtraction

$$\begin{aligned}(G_1\,S_1\ +\ \ldots\ G_p\,S_p\ +\ G)-\ \left(G'_1\,S_1+\ldots G'_p\,S_p\ +\ G'\right)\\ =\ (G_1-G'_1)\,S_1+\ldots(G_p-G'_p)\ S_p\ +(G-G')\end{aligned}\tag{71}$$

5. Simultaneous substitution. Here is the result of one symbol being substituted:

$$\begin{aligned}Subst(\,G_1\,S_1\ +\ \ldots\ G_p\,S_p\ +\ G;\ S_1\ \rightarrow\ H_1S_1+\ldots H_pS_p+\ H\\ =\ G_1\left(H_1S_1+\ldots H_pS_p+\ H\right)+\ G_2\,S_2+\cdots\ G_p\,S_p\ +\ G\end{aligned}\tag{72}$$

Any $V\colon \mathcal{S}\ \rightarrow \mathcal{I}$ can be extended to $V\colon \mathcal{A}\ \rightarrow \mathcal{I}$ by

$$V\left(G_1\,S_1\ +\ \ldots\ G_p\,S_p\ +\ G\right)=\ G_1*V(S_1)+\ldots+G_p*V(S_p)\ +\ G\tag{73}$$

We will call such a V, a value function. Notice that for any value function V and any interval G, $V(G)=G$.

Lemma 1. If V associates with each symbol a singleton interval $V(S)=[s,s]$ then for any $[A,A']\ \in\ \mathcal{A}$, and $J\in\mathcal{I}$.

$$a\in\ V(A),a'\in V(A')\Rightarrow a+a'\ \in\ V(A\ +\ A')\tag{74}$$

$$a\in\ V(A),a'\in V(A')\Rightarrow a-a'\ \in\ V(A-\ A')\tag{75}$$

$$a\in\ V(A),i\in\mathcal{I}\ \Rightarrow ai\ \in\ V(A*I)\tag{76}$$

$$a\in\ V(A),i\in\mathcal{I}\ \Rightarrow a/i\ \in\ V(A/I)\tag{77}$$

$$a\in\ V(A),s_i\in V(A')\ \Rightarrow a\in\ V(Subst[A,S_i\rightarrow A')\tag{78}$$

Proof: Let

$$A\ =\ G_1\,S_1\ +\ \ldots\ G_p\,S_p\ +\ G\tag{79}$$

$$A'\ =\ G'_1\,S_1\ +\ \ldots\ G'_p\,S_p\ +\ G'\tag{80}$$

$$a\ =\ g_1s_1+\ldots+g_ps_p\ +\ g\quad\text{with}\ g_i\in G_i,g\in G\tag{81}$$

$$a'\ =\ g'_1s_1+\ldots+g'_ps_p\ +\ g'\quad\text{with}\ g'_i\in G_i,g'\in G\tag{82}$$

Then, using distributivity, we get

$$\begin{aligned} a + a' &= (g_1 1 + g'_1)s_1 + \ldots + (g_p + g'_p)s_p + (g + g') \\ &\in (G_1 + G'_1)S_1 + \ldots + (G_p + G'_p)S_p + (G + G') \\ &\in V(A + A) \end{aligned} \tag{83}$$

The other cases are similar. Notice that the first two implications do not necessarily hold if $V(S)$ is not a singleton interval as interval multiplication is not distributive over interval addition. QED

4.1.7.2. Error Set Abstractions

For each variable x, we will define $E(x) \in \mathcal{A}$ and then show how it represents an abstraction of the set of errors for variable x. More precisely, $E(x)$ will represent a symbolic combination of the inputs and outputs.

Definition of $E(x)$. For constants c, let $E(c)$ be a constant interval that includes all possible error values for c.

For any input u_i, let $U_i \in \mathcal{S}$ and let $E(u_i) = U_i = [1,1]\ U_i \in \mathcal{A}$.

Similarly, for output y_i, let $Y_i \in \mathcal{S}$ and $E(y_i) = Y_i = [1,1] Y_i \in \mathcal{A}$.

Suppose that for all x that were assigned a value before v in the filter code, $E(x) \in \mathcal{A}$ was defined. Then the definition for $E(v)$ depends on the statement that assigns it a value.

$v = w$:

$$E(v) = E(w) \tag{84}$$

$v = c$:

$$E(v) = E(c) \tag{85}$$

$v = w + z$:

$$E(v) = R(v) * [-\epsilon_m, \epsilon_m] + (E(w) + E(z)) * [1 - \epsilon_m, 1 + \epsilon_m] \tag{86}$$

$v = w - z$:

$$E(v) = R(v) * [-\epsilon_m, \epsilon_m] + (E(w) - E(z)) * [1 - \epsilon_m, 1 + \epsilon_m] \tag{87}$$

$v = c * w$:

$$E(v) = R(v) * [-\epsilon_m, \epsilon_m] + \left(E(c) * R(w) + (R(c) + E(c)) * E(w)\right) * [1 - \epsilon_m, 1 + \epsilon_m] \tag{88}$$

$v = w / c$:

$$E(v) = \frac{-R(v)\,E(c) + R(w) * [-\epsilon_m, \epsilon_m] + E(w) * [1 - \epsilon_m, 1 + \epsilon_m]}{R(c) + E(c)} \tag{89}$$

Lemma 2. Let $\epsilon_i(u)$ and $\epsilon_i(y)$ be the error for input *i* and output *i* respectively and V the value function with $V(U_i) = [\epsilon_i(u), \epsilon_i(u)]$ and $V(Y_i) = [\epsilon_i(y), \epsilon_i(y)]$. Then for any variable x

$$\forall\, n \geq 0.\, \epsilon_i(x) \in V\big(E(x)\big). \tag{90}$$

Proof: From the definition of $E(x)$, the statement is trivially true for the state variables and constants. Assume that the statement is true for the variables given a value before v. Then the first two cases are obvious.

Case $v = w + z$:

$$\begin{aligned} & f_n(v) - r_n(v) \\ &= (f_n(w) + f_n(z))(1+\delta) - (r_n(w) + r_n(z)) \quad \text{where } |\delta| \leq \epsilon_m \\ &= (r_n(w) + \epsilon_n(w)) + (r_n(z) + \epsilon_n(z)))(1+\delta) - (r_n(w) + r_n(z)) \\ &= (r_n(w) + r_n(z))\,\delta + (\epsilon_n(w) + \epsilon_n(z))(1+\delta) \\ &= r_n(v)\,\delta + (\epsilon_n(w) + \epsilon_n(z))(1+\delta) \end{aligned} \tag{91}$$

then,

$$\begin{aligned} & r_n(v) \in R(v) = V\big(R(v)\big), \\ & \delta \in [-\epsilon_m, \epsilon_m] = V([-\epsilon_m, \epsilon_m]), \\ & 1+\delta \in [1-\epsilon_m, 1+\epsilon_m] = V([1-\epsilon_m, 1+\epsilon_m]\ ) \end{aligned}$$

and from the inductive hypothesis we have, $\epsilon_n(w) \in V(E(w))$ and $\epsilon_n(z) \in V(E(z))$. Therefore, by Lemma 1,

$$\begin{aligned} f_n(v) - r_n(v) \in V(R(v) * [-\epsilon_m, \epsilon_m] + (E(w) + E(z)) * [1-\epsilon_m, 1+\epsilon_m]) \\ = V\big(E(v)\big). \end{aligned} \tag{92}$$

Case $v = w - z$: similar to previous case

Case $v = c * w$:

$$\begin{aligned} & f_n(v) - r_n(v) \\ &= f(c)\, f_n(w)\,(1+\delta) - r(c)\, r_n(w) \quad \text{where } |\delta| \leq \epsilon_m \\ &= (r(c) + \epsilon(c))\,(r_n(w) + \epsilon_n(w))\,(1+\delta) - r(c)\, r_n(w) \\ &= r(c)\, r_n(w)\ \delta + (\epsilon(c) r_n(w) + r(c)\,\epsilon_n(w) + \epsilon(c)\,\epsilon_n(w))\,(1+\delta) \\ &= r_n(v)\ \delta + (\epsilon(c) r_n(w) + \big(r(c) + \epsilon(c)\big)\,\epsilon_n(w))\,(1+\delta) \\ &\in V(R(v) * [-\epsilon_m, \epsilon_m] + (E(c) * R(w) + \big(R(c) + E(c)\big) * E(w)) * [1-\epsilon_m, 1+\epsilon_m]\ ) \\ &= V(E(v)) \end{aligned}$$

Case $v = w / c$:

$$f_n(v) - r_n(v)$$

$$= \ (f_n(w) \ / \ f(c)) \ (1 + \delta) \ - \ r_n(w) \ / \ r(c) \quad \text{where} \ \ |\delta| \leq \epsilon_m$$

$$= \ \frac{(r_n(w) \ + \ \epsilon_n(w)) \ (1 + \delta)}{r(c) \ + \ \epsilon(c)} \ - \ \frac{r_n(w)}{r(c)}$$

$$= \ \frac{(r_n(w) \ + \ \epsilon_n(w)) \ (1 + \delta) \ - \ r_n(w) \ (r(c) \ + \ \epsilon(c)) \ / \ r(c)}{r(c) \ + \ \epsilon(c)} \tag{93}$$

$$= \ \frac{(r_n(w) \ r(c) \ - \ r_n(w) \ r(c) \ - \ r_n(w) \ \epsilon(c)) \ / \ r(c) \ + \ r_n(w) \ \delta + \ \epsilon_n(w) \ (1 \ + \ \delta)}{r(c) \ + \ \epsilon(c)}$$

$$= \ \frac{- \ r_n(w) \ \epsilon(c) \ / \ r(c) \ + \ r_n(w) \ \delta + \ \epsilon_n(w) \ (1 \ + \ \delta)}{r(c) \ + \ \epsilon(c)}$$

$$\in V\left(\frac{- \ R(v) * E(c) \ + \ R(w) * [-\epsilon_m, \epsilon_m] \ + \ E(w) * [1 - \epsilon_m, 1 + \epsilon_m]}{R(c) \ + \ E(c)}\right)$$

$$= \ V\big(E(v)\big).$$

QED

Notice that, for any $x, E(x)$ is either a symbol (in the case when *x* is an input or a past output), a constant interval (in the case **x** is a constant value) or is defined as a linear expression over other error set abstractions. Therefore any $E(x)$ can be expressed as a linear combination of input symbols U_i and past output symbols Y_i.

Suppose that for the output variable y, $E(y) \in \mathcal{A}$ is

$$E(y) = \ A_0 \ U_{n+1} + \ldots + A_N \ U_{n-N+1} \ + B_1 \ Y_n + \ldots + B_N \ Y_{n-N+1} + \ P \tag{94}$$

Without loss of generality we can assume P is symmetric about 0. Indeed, we can always enlarge P into a symmetric interval and the effect will be a larger $E(y)$. We can treat P similar to an input error at iteration $n + 1$ by introducing a symbolic variable P_{n+1}.

$$E(y) = \ A_0 \ U_{n+1} + \ldots + A_N \ U_{n-N+1} \ + B_1 \ Y_n + \ldots + B_N \ Y_{n-N+1} + [1,1] \ P_{n+1} \tag{95}$$

We will define recursively $E_n(y) \in \mathcal{A}$ to be the unfolded version of $E(y)$ of "depth" n.

$$E_k(y) = [0,0] \quad \text{if} \ k = 0, \ldots, N-1 \tag{96}$$

$$E_{k+1}(y) \ = \ Subst(E(y), Y_n \to E_k(y), \ldots, Y_{n-N+1} \to E_{k-N+1}(y)) \quad \text{for} \ k \geq N \tag{97}$$

With these definitions,

$$E_{n+1}(y) = \ C_0 \ U_{n+1} + \ldots + C_N \ U_1 \ + D_0 \ P_{n+1} + \ldots + D_n P_1 \tag{98}$$

where C_n, D_n are intervals and for $n \ > \ N$,

$$C_{n+1} \ = \ B_1 C_n + \ldots + B_N C_{n-N+1} \tag{99}$$

$$D_{n+1} \ = \ B_1 P_n + \ldots + B_N P_{n-N+1} \tag{100}$$

4.1.7.3. Error Bounds for First-Order Linear Filters

For a first order filter, the recurrence equations are

$$C_0 = A_0 \tag{101}$$

$$C_1 = A_1 + B_1 A_0 \tag{102}$$

$$C_{n+1} = B_1 C_n \tag{103}$$

$$D_0 = [1,1] \tag{104}$$

$$D_1 = B_1[1,1] = B_1 \tag{105}$$

$$D_{n+1} = B_1 D_n \tag{106}$$

which have the solution

$$C_{n+1} = C_1 B_1^n \tag{107}$$

$$D_{n+1} = D_1 B_1^n = B_1^{n+1} \tag{108}$$

Let V the value function for which $V(U_i) = EV(u_i)$ (the set of error values for u_i) and $V(P_i) = P$. Then

$$\begin{aligned} V(E_{n+1}(y)) = {} & C_0\, EV(u_{n+1}) + C_1\, EV(u_n) + \ldots C_1 B_1^n EV(u_1) + \\ & D_0\, EV(p_{n+1}) + D_1\, EV(p_n) + \ldots D_1 B_1^n EV(p_1) \end{aligned} \tag{109}$$

For an interval $I = [a, b]$, let $max\ |I| = \max\{\, |x| \mid x \in [a,b]\} = max(|a|, |b|)$.

As $EV(u)$ and P are symmetric about 0,

$$\begin{aligned} max\ |V(E_{n+1}(y))| \leq {} & max\ |C_0|\ max(EV(u)) + max(P) + \\ & max\ |C_1|\ max(EV(u)) + \ldots + max\ |C_1|\ (max\ |B_1|)^n\ max(EV(u)) + \\ & max\ |D1|\ max(P)(1 + \ldots + (max\ |B_1|)^n) \\ = {} & max\ |C_0|\ max(EV(u)) + max(P) + \\ & (max\ |C_1|\ max(EV(u)) + max\ |B_1|\ max(P))(1 + \ldots + (max\ |B_1|)^n) \end{aligned}$$

For example, let's reconsider the lead lag filter described by

$$y_0 = 0 \tag{110}$$

$$y_{n+1} = \frac{9}{13} u_{n+1} - \frac{7}{13} u_n + \frac{11}{13} y_n \tag{111}$$

with $u_0 = 0$ and for any $i > 0,\ u_i \in [-1, 1]$.

The analysis produces the following recurrence equation for the accumulated error after $n + 1$ iterations

$$E_{n+1}(y) = A_0\, EV(u_{n+1}) + A_1 EV(u_n) + B_1 EV(y_n) + EV(p_{n+1}) \quad (112)$$

where

$$A_0 = \begin{bmatrix} 0.692307207006099942128200016249 50, \\ 0.692308166323494155612921237581 08 \end{bmatrix} \quad (113)$$

$$A_1 = [-0.538461956046894423538365413644 70, \\ -0.538461177306850481038353622644 72] \quad (114)$$

$$B_1 = [0.846153374254430682696991980034 44, \\ 0.846154385770028390403414859247 19] \quad (115)$$

$$EV(p_n) = P = [-0.000000733595920042139677812438 23, \\ 0.000000733595920042139677812438 23] \quad (116)$$

Therefore,

$$\left|V\left(E_{n+1}(y)\right)\right| \leq A_{error} + M_{error}(1 + \ldots + \beta^n) \quad (117)$$

where

$$C_0 = A_0 = [0.692307207006099942128200016249 50, \\ 0.692308166323494155612921237581 08] \quad (118)$$

$$C_1 = A_1 + B_1 C_0 = [0.047336123181977676389225528090 80, \\ 0.047338413932180370232611870488 24] \quad (119)$$

$A_{error} = max|C_0| max\left(EV(u)\right) + max(P) = 0.000000816125484699767517331571 94$
$M_{error} = max|C_1| max\left(EV(u)\right) + max|B_1| max(P) = 0.000000626378583819971585633813 00$
β = max$|B_1|$ = 0.846154385770028390403414859247 19

4.1.7.4. Error Bounds for Second-Order Linear Filters

For a second-order filter, the recurrence equations are

$$C_0 = A_0 \quad (120)$$

$$C_1 = A_1 + B_1 C_0 \quad (121)$$

$$C_2 = A_2 + B_1 C_1 + B_2 C_0 \quad (122)$$

$$C_{n+1} = B_1 C_n + B_2 C_{n-1} \quad (123)$$

$$D_0 = [1,1] \quad (124)$$

$$D_1 = B_1[1,1] = B_1 \quad (125)$$

$$D_2 = B_1 D_1 + B_2[1,1] = B_1^2 + B_2 \quad (126)$$

$$D_{n+1} = B_1 D_n + B_2 D_{n-1} \quad (127)$$

Consider all the solutions $r_1 < r_2$ of the equation

$$x^2 = b_1 x + b_2 \tag{128}$$

with $b_1 \in B_1$ and $b_2 \in B_2$. Then

$$R_1 = \frac{B_1 - \sqrt{B_1^2 + 4B_2}}{2} \tag{129}$$

$$R_2 = \frac{B_1 + \sqrt{B_1^2 + 4B_2}}{2} \tag{130}$$

and therefore

$$\{r_1 \mid r_1 \text{ the smaller solution of } x^2 = b_1 x + b_2 \text{ for } b_i \in B_i\} \subseteq R_1$$
$$\{r_2 \mid r_2 \text{ the larger solution of } x^2 = b_1 x + b_2 \text{ for } b_i \in B_i\} \subseteq R_2$$

In the case when $R_1$and R_2 do not intersect, we can define a few constants that are needed in what follows

$$\beta_1 = \frac{max(R_1)\, max(R_2 - R_1) + max(R_2)\, max(R_2 - R_2)}{min(R_2 - R_1)} \tag{131}$$

$$\beta_2 = \frac{max(R_1)\, max(R_1 - R_1) + max(R_2)\, max(R_2 - R_1)}{min(R_2 - R_1)} \tag{132}$$

$$\beta = max\,(\beta_1, \beta_2) \tag{133}$$

$$M = max\left(\frac{max\,|C_1 R_2 - C_2|}{min(R_2 - R_1)}, \frac{max\,|C_2 - C_1 R_1|}{min(R_2 - R_1)}\right) \tag{134}$$

$$N = max\left(\frac{max\,|D_1 R_2 - D_2|}{min(R_2 - R_1)}, \frac{max\,|D_2 - D_1 R_1|}{min(R_2 - R_1)}\right) \tag{135}$$

Theorem: If $0 < min(R_1)$ and $max(R_1) < min(R_2)$ then for any $n > 1$,

$$|E_{n+2}(y)| \le [max\,|C_0| + max\,|C_1| + max\,|C_2| + 2M max(R_2)\,(1 + \ldots + \beta^{n-1})]\, max(EV(u)) + [max\,|D_0| + max\,|D_1| + max\,|D_2| + 2\,N\, max(R_2)(1 + \ldots + \beta^{n-1})]\, max(P) \tag{136}$$

The theorem implies that

$$max\,|V(E_{n+1}(y))| \le A_{error} + M_{error}\,(1 + \cdots + \beta^n) \tag{137}$$

where

$$A_{error} = [\max|C_0| + \max|C_1| + \max|C_2|]\, \max\big(EV(u)\big) + [max\,|D_0| + max\,|D_1| + max\,|D_2|]\, \max(\mathrm{P}) \tag{138}$$

$$M_{error} = 2\, M max(R_2)\, max(EV(u)) + 2\, N\, max(R_2)\, max(P) \tag{139}$$

Notations: Let $c_0, c_1, \ldots$ be such that $c_0 \in C_0, c_1 \in C_1$ and $\forall\, n > 0. c_{n+1} = b_{n+1,1}\, c_n + b_{n+1,2}\, c_{n-1}$ with $b_{n+1,i} \in B_i$. For each $n > 0$, let $r_{n+1,1} < r_{n+1,2}$ be the solutions of

$$x^2 = b_{n+1,1} x + b_{n+1,2} \tag{140}$$

and let s_{n+1} and t_{n+1} be the solutions of

$$c_{n-1} = s_{n+1} + t_{n+1} \tag{141}$$

$$c_n = s_{n+1}\, r_{n+1,1} + t_{n+1}\, r_{n+1,2} \tag{142}$$

Lemma 3: If $0 < min(R_1)$ and $max(R_1) < min(R_2)$ then

$max\ (|s_3|, |t_3|) \leq M$

$max\ (|s_{n+2}|, |t_{n+2}|) \leq max\ (|s_{n+1}|, |t_{n+1}|) * \beta$

Proof: s_3 and t_3 are the solutions of the system

$$c_1 = s_3 + t_3 \tag{143}$$

$$c_2 = s_3\, r_{3,1} + t_3 r_{3,2} \tag{144}$$

Therefore

$$s_3 = \frac{c_1 r_{3,2} - c_2}{r_{3,2} - r_{3,1}} \tag{145}$$

$$t_3 = \frac{c_2 - c_1 r_{3,1}}{r_{3,2} - r_{3,1}} \tag{146}$$

which imply the first inequality of Lemma 3. Now, for any $n > 1$,

$$\begin{aligned} c_{n+1} = b_{n+1,1}\, c_n + b_{n+1,2}\, c_{n-1} &= b_{n+1,1}(s_{n+1}\, r_{n+1,1} + t_{n+1}\, r_{n+1,2}) + b_{n+1,2}\, (s_{n+1} + t_{n+1}) \\ &= s_{n+1}\, (b_{n+1,1}\, r_{n+1,1} + b_{n+1,2}) + t_{n+1}(b_{n+1,1}\, r_{n+1,2} + b_{n+1,2}) \\ &= s_{n+1}\, r_{n+1,1}^2 + t_{n+1}\, r_{n+1,2}^2 \end{aligned} \tag{147}$$

From the definitions of $r_{n+1,i}$ and $r_{n+2,i}$,

$$c_n = s_{n+1}\, r_{n+1,1} + t_{n+1}\, r_{n+1,2} = s_{n+2} + t_{n+2} \tag{148}$$

$$c_{n+1} = s_{n+1}\, r_{n+1,1}^2 + t_{n+1}\, r_{n+1,2}^2 = s_{n+2}\, r_{n+2,1} + t_{n+2}\, r_{n+2,1} \tag{149}$$

and by solving this system for s_{n+2} and t_{n+2} we get

$$\forall n > 0. s_{n+2} = s_{n+1} \frac{r_{n+1,1}(r_{n+2,2} - r_{n+1,1})}{r_{n+2,2} - r_{n+2,1}} + t_{n+1} \frac{r_{n+1,2}(r_{n+2,2} - r_{n+1,2})}{r_{n+2,2} - r_{n+2,1}} \quad (150)$$

$$\forall n > 0. t_{n+2} = s_{n+1} \frac{r_{n+1,1}(r_{n+1,1} - r_{n+2,1})}{r_{n+2,2} - r_{n+2,1}} + t_{n+1} \frac{r_{n+1,2}(r_{n+1,2} - r_{n+2,1})}{r_{n+2,2} - r_{n+2,1}} \quad (151)$$

therefore

$$|s_{n+2}| \leq |s_{n+1}| \frac{max(R_1)\, max(R_2 - R_1)}{min(R_2 - R_1)} + |t_{n+1}| \frac{max(R_2)\, max(R_2 - R_2)}{min(R_2 - R_1)} \quad (152)$$

$$|t_{n+2}| \leq |s_{n+2}| \frac{max(R_1)\, max(R_1 - R_1)}{min(R_2 - R_1)} + |t_{n+1}| \frac{max(R_2)\, max(R_2 - R_1)}{min(R_2 - R_1)} \quad (153)$$

and thus

$$|s_{n+2}| \leq max\,(|s_{n+1}|, |t_{n+1}|)\, \beta_1 \quad (154)$$

$$|t_{n+2}| \leq max\,(|s_{n+1}|, |t_{n+1}|)\, \beta_2 \quad (155)$$

which imply the second inequality of Lemma 3. QED

Proof of Theorem: From Lemma 3,

$$max\,(|s_{n+2}|, |t_{n+2}|) \leq max\,(|s_{n+1}|, |t_{n+1}|)\beta \leq M\, \beta^{n-1} \quad (156)$$

Thus

$$|c_{n+2}| \leq M\beta^{n-1}\, r_1 + M\beta^{n-1}\, r_2 < 2Mmax(R_2)\, \beta^{n-1} \quad (157)$$

which implies

$$max\,|C_{n+2}| \leq 2Mmax(R_2)\, \beta^{n-1} \quad (158)$$

Similarly, for the sequence D_n,

$$max\,|D_{n+2}| \leq 2Nmax(R_2)\, \beta^{n-1} \quad (159)$$

Thus

$$E_{n+2}(y) \in C_0\, EV(u_{n+1}) + \ldots + C_n\, EV(u_1) + D_0\, EV(p_{n+1}) + \ldots + D_n\, EV(p_1)$$

$$|E_{n+2}(y)| \leq max\,|\,C_0|\,max(EV(u)) + \ldots + max\,|\,C_n|\,max(EV(u)) + max\,|\,D_0|max(P) + \ldots + max\,|\,D_n|max(P) \quad (160)$$

$$= (max|C_0| + max|C_1| + max|C_2| + 2Mmax(R_2)(1 + \cdots + \beta^{n-1})\,max(EV(u)) + max|D_0| + max|D_1| + max|D_2| + 2Nmax(R_2)\,(1 + \cdots + \beta^{n-1})\,max(P)$$

QED

Example: consider again the second-order filter (from Section 4.2.4)

$$y_0 = y_1 = 0 \quad (161)$$

(162)

$$y_{n+1} = \frac{592841}{78010601}u_{n+1} + \frac{1185682}{78010601}u_n + \frac{592841}{78010601}u_{n-1} + \frac{126814318}{78010601}y_n - \frac{51175081}{78010601}y_{n-1}$$

where $u_0 = u_1 = 0$ and for any $i > 0,\ u_i \in [-100, 100]$.

The analysis produces the following recurrence equation for the accumulated error of the output

$$E_{n+1}(y) = A_0EV(u_{n+1}) + A_1EV(u_n) + A_2EV(u_{n-1}) + B_1EV(yn) + B_2EV(y_{n-1}) + EV(p_{n+1}) \quad (163)$$

where

A_0 = [0.007599487774249052183158020118 99,
0.007599500782683565976490134765 43]

A_1 = [0.015198979172214314526012967498 56,
0.015199005189091704921039141108 64]

A_2 = [0.007599488680178698696464959299 48,
0.007599499876752584727564838987 53]

B_1 = [1.625602960513804717132181940621 87,
1.625604584042258058926800631978 86]

B_2 = [-0.656001827475824275144528840151 65,
-0.656001287357407015482649345624 27]

P = [-0.000039589480719188901180420995 20,
0.000039589480719188901180420995 20]

$C_0 = A_0$ = [0.007599487774249052183158020118 99,
0.007599500782683565976490134765 43]

$C_1 = A_1 + B_1C_0$ = [0.027552728996422038198346616337 96,
0.027552788497854837745776850068 90]

$C_2 = A_2 + B_1C_1 + B_2C_0$
= [0.047404000105652539334323323639 06,
0.047404165398847956558342881171 12]

D_0 = [1,1]

$D_1 = B_1\ [1,1] = B_1 = [1.625602960513804717132181940621 87,$
$1.625604584042258058926800631978 86]$

$D_2 = B_1\ D_1 + B_2\ \ [1,1] = B_1^2 + B_2$
$= [1.986583157755422263183541795062 56,$
$1.986588976301795829123834853517 86]$

$$R_1 = \frac{B_1 - \sqrt{B_1^2 + 4\,B_2}}{2} \tag{164}$$

$= [0.744637864174760005215702854550 46,$
$0.744652318932452322066720494728 25]$

$$R_2 = \frac{B_1 + \sqrt{B_1^2 + 4\,B_2}}{2} \tag{165}$$

$= [0.880951453345579065962770791572 11,$
$0.880965908103271382813788431749 91]$

$$\beta_1 = \frac{max(R_1)\,max(R_2 - R_1) + max(R_2)\,max(R_2 - R_2)}{min(R_2 - R_1)} \tag{166}$$

$$= 0.744903690201644372551039556225 01$$

$$\beta_2 = \frac{max(R_1)\,max(R_1 - R_1) + max(R_2)\,max(R_2 - R_1)}{min(R_2 - R_1)} \tag{167}$$

$$= 0.881231735663107671254432673511 80$$

$$\beta = max\,(\beta_1, \beta_2) = 0.881221240508504514327519934077 94 \tag{168}$$

$$M = max(\frac{max\,|C1\,R2 - C2|}{min(R2 - R1)}, \frac{max\,|C2 - C1\,R1|}{min(R2 - R1)}) \tag{169}$$

$$= 0.197267284510185372137209938667 27$$

$$N = max(\frac{max\,|D1\,R2 - D2|\}\{\}}{min(R2 - R1)}, \frac{max\,|D2 - D1\,R1|}{min(R2 - R1)}) \tag{170}$$

$$= 5.694118771409338615442925531299 93$$

$$A_{error} = 0.000183578497661424225042079733 51 \tag{171}$$

$$M_{error} = 0.000401330740496774269394088187 60 \tag{172}$$

4.1.8. Accumulating Integrators: Error Range Functions

In contrast to digital filters, accumulating integrators have no discounting mechanism for past inputs. Instead, the output range intentionally grows without bound. An analysis of the floating-point errors can only give bounds as a function of the number of iterations. The analyst can then decide how many iterations to allow before the error becomes unacceptable.
Let

$$x_0 = 0 \tag{173}$$

$$x_{n+1} = x_n + c \tag{174}$$

be the recursive equation of an integrator. In this case, we are interested in the range of values after at most n iterations, which is $[0, nc]$ or $[nc, 0]$ depending whether c is a positive or negative constant.

To find the error range after n iterations or less, we cannot use the same expressions as for the analysis of filters. In that case, we used the fact that all the variables involved have finite ranges which we computed in advance. However, we can adapt the method by changing the symbolic interval combinations used for filters with symbolic combinations over linear expressions in n, the number of iterations.

The real value after n iterations is $R_n(x) = nc$. In the simple case when c is found to be representable then the error found is 0. Otherwise, the analysis finds the following recurrence relation.

$$E_{n+1}(x) = E_n(x)\, B_1 + Q_n + P \tag{175}$$

$$B_1 = [1 - \epsilon_m, 1 + \epsilon_m] \tag{176}$$

$$Q = c * [-\epsilon_m, \epsilon_m] \tag{177}$$

$$P = E(c) * [1 - \epsilon_m, 1 + \epsilon_m] \tag{178}$$

from which we get the closed form

$$|V(E_{n+1}(x))| \leq max\,|Q|\,(n + (n-1)\beta + \cdots + \beta^n) + max\,|P|\,(1 + b\beta + \cdots + \beta^n) \tag{179}$$

where $\beta = 1 + \epsilon_m$. Notice that

$$\begin{aligned} n + (n-1)\beta + \cdots + \beta^n \\ &= (1 + b\beta + \cdots + \beta^n) + (1 + b\beta + \cdots + \beta^{n-1}) + \ldots + 1 \\ &= \frac{\beta^{n+1} - 1}{\beta - 1} + \frac{\beta^n - 1}{\beta - 1} + \cdots + \frac{\beta - 1}{\beta - 1} \\ &= \frac{\beta + \cdots + \beta^{n+1} - n}{\beta - 1} \\ &= \frac{\beta}{\beta - 1}(1 + \cdots + \beta^n) - \frac{n}{\beta - 1} \end{aligned} \tag{180}$$

and therefore

$$\left|V\left(E_{n+1}(x)\right)\right| \leq max|Q|\frac{\beta}{\beta-1}(1+\beta+\cdots+\beta^n) - max|Q|\frac{n}{\beta-1} + max\,|P|\,(1+\beta+\cdots+\beta^n)$$

or

$$max\,|V(E_{n+1}(y))| \leq A_{error} + M_{error}(1+\beta+\cdots+\beta^n) + N_{error}n$$

where

$$A_{error} = 0$$

$$M_{error} = max|Q|\frac{\beta}{\beta-1} + max\,|P| = (|c| + E(c))(1+\epsilon_m)$$

$$N_{error} = -\frac{max\,|Q|}{\beta-1}$$

Example: Consider the integrator described by

$$x_0 = 0 \tag{181}$$

$$x_{n+1} = x_n + 0.1 \tag{182}$$

with the output variable given by $y_n = x_n * 3.2$. This is the essence of the integrator that caused the Patriot Missile failure. The error bound for the integrator is then

$$max\,|V(E_{n+1}(x))| \leq A_{x,error} + M_{x,error}(1 + \cdots + \beta^n) + N_{x,error}\,n$$

where

$$A_{x,error} = 0$$

$$M_{x,error} = (|c| + E(c))(1+\epsilon_m) = 0.10000001788139414315992326010019$$

$$N_{x,error} = -|c| = -0.1$$

The error bound on the output is

$$E_n(y) = R_n(y) * [-\epsilon_m, \epsilon_m] + [E(3.2) * R_n(x) + (3.2 + E(3.2)) * E_n(x)] * [1-\epsilon_m, 1+\epsilon_m] \tag{183}$$

and thus

$$max\,|V(E_{n+1}(y))| \leq 0.32\,\epsilon_m n + [E(3.2) * 0.1\,n + \left(3.2 + E(3.2)\right) * (M_{x,error}(1 + \cdots + \beta^n) - 0.1\,n)]\,(1+\epsilon_m) = A_{y,error} + M_{y,error}\,(1 + \cdots + \beta^n) + N_{y,error}\,n \tag{184}$$

where

$$
\begin{aligned}
A_{y,error} &= 0 \\
M_{y,error} &= \big(3.2 + E(3.2)\big)(1+\epsilon_m)M_{x,error} \\
&= 0.32000011444093274803971696172939 \qquad (185) \\
N_{y,error} &= 0.32\,\epsilon_m + 0.1\,E(3.2)(1+\epsilon_m) - \big(3.2 + E(3.2)\big)0.1\,(1+\epsilon_m) \\
&= 0.32\,\epsilon_m - 0.32\,(1+\epsilon_m) = -0.32
\end{aligned}
$$

4.2. Summary of Filter Test Suite Results

The development of the first year analysis of Linear Digital Filters was guided by, tuned, and ultimately tested for round trip integration by a suite of test cases designed by the Honeywell team. These consisted of Simulink models of representative filter applications, their generated C implementations and their generated JSON files describing model-level properties of the generated code to be exploited by CodeHawk. This section summarizes the results of each of these test cases.

General Significance and Organization of the Results.

As discussed in Section 3.1, the goal of the combined model-level and code-level analysis is to derive precise, consistent bounds for the range and numerical error of the filter outputs. Both the range and error need to be bounded in order to guarantee stability of the control algorithm. As we have pointed out earlier in this report, most static analysis tools are not able to compute such a bound and can thus report high bounds. This forces current practices to rely upon informal, empirical manual analysis/reviews and simulations of the code. Our results demonstrate that our combined analysis technique proves precise, conservative bounds for both variable range and numerical error – representing a definite advancement in the state-of-art of static analysis approaches for numerical algorithms.

Range Bounds Results: The range at the output of the filter is theoretically expected to be bounded. The bounds should be stable over time – often proportionally related to the input range bound based upon the filter coefficients which are derived from the time constants (T_n and T_d) of the filter and the sample period (T_s). As presented in the following subsections, the results returned by the CodeHawk analysis of filter range bounds on he source code confirm the theoretically expected properties. Furthermore the bounds are precise (tight), yet conservative – i.e., representing the worst case if value at the filter input is varied within its range arbitrarily over time.

Numerical Error Bounds Results: For a properly designed filter, the numerical error at the output of the filter is expected to be bounded to a small value (less than 1% of the range) in order to achieve control stability. The error bounds should be convergent over time. The ideal *dynamic* scenario is that the numerical error should approach 0 as time approaches infinity if the filter input is held at a constant value. This ideal dynamic scenario, however, is hard to achieve in a *static* analysis framework. Thus, industry practice relies on empirically testing the tolerance for error to be within 1% of the variable value. Our analysis proves the error bound to be well within

this tolerance even for 2nd order (quadratic) filter; furthermore the error bound remains constant with time.

4.2.1. Filter Example using a Resettable Lead Lag Filter

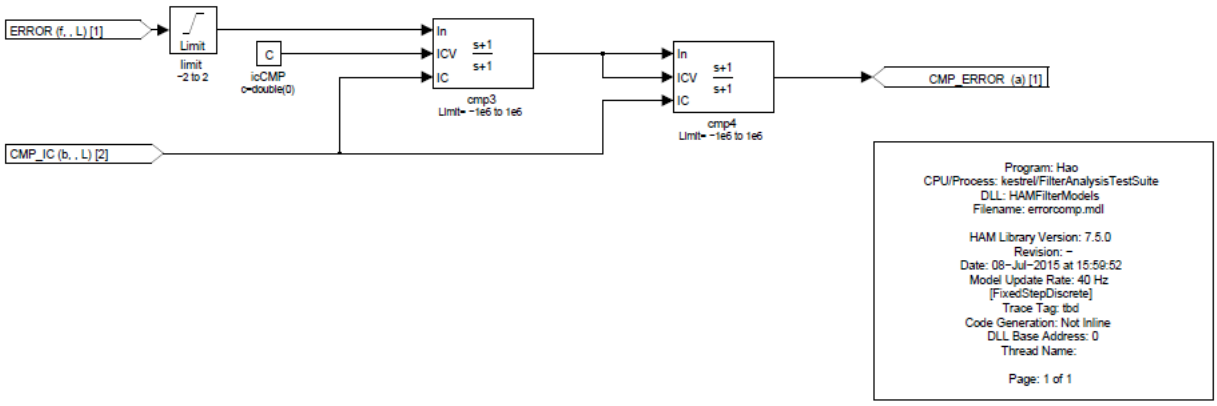

Figure 13 Resettable Lead Lag Filter - Test Model Diagram

Figure 13 shows 'errorcomp', a model with two resettable lead lag filters in series, creating a 2nd order filter. The next subsection describes the transfer function and difference equation derivation for this class of filters. Following subsection shows the analysis results assuming the filter runs for 72,000 seconds at 40 Hz (25 ms period).

4.2.1.1. Resettable Lead Lag Filter Transfer Function

Figure 14 shows the Resettable Lead Lag filter as it appears as a block in MATLAB Simulink. This class of filters has two time constants, Tn and Td, that are both constants for a particular filter instance. There are additional *reset* (IC) and *reset value* (ICV) inputs that allow the filter to be reset.

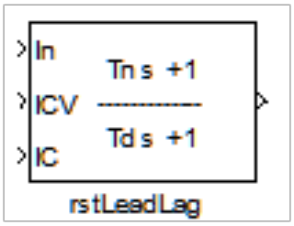

Figure 14 Resettable Lead Lag Filter

The following is the transfer function derivation for this filter class:

Reset Mode (when $IC = 1$): $y = ICV$

Filter Mode Transfer Function (when $IC = 0$):

General form:

Continuous domain: $H(s) = \frac{T_n s+1}{T_d s+1}$ **Discrete domain:** $H(z) = \frac{k_3+k_4 z^{-1}}{k_1+k_2 z^{-1}}$

Coefficient definitions (all coefficients are constant)

$$k_1 = 1 + \frac{2T_d}{T_s} \qquad k_2 = 1 - \frac{2T_d}{T_s} \qquad k_3 = 1 + \frac{2T_n}{T_s} \qquad k_4 = 1 - \frac{2T_n}{T_s}$$

Difference equation:

$$y_n = (k_3 u_n + k_4 u_{n-1} - k_2 y_{n-1})/k_1 \tag{186}$$

4.2.1.2. Analysis results after 72,000 Seconds (20 hrs)

System Oper Time (secs): 72000 Model Period (ms): 25 Total Periods: 2880000						
# Symbol Name	Normal Range Min	Normal Range Max	Range Constraint Min (if any)	Range Constraint Max (if any)	Data Type	Worst Case Numerical Error Bounds
CMP_ERROR	-11.65675964	11.65675964	-11.65675964	11.65675964	LangIndep_Float32	eB[-+1.33e-003]
# Filters						
cmp3	-3.950617284	3.950617284	-3.950617284	3.950617284	LangIndep_Float32	eB[-+2.29e-004]
cmp4	-11.65675964	11.65675964	-11.65675964	11.65675964	LangIndep_Float32	eB[-+1.33e-003]

Figure 15 Resettable Lead Lag Filter - 72000 Seconds

In Figure 15, variable CMP_ERROR shows the analysis for the final output of the model in Figure 13 for a run time of 72000 seconds. Cmp3 and cmp4 show the analysis for the outputs of filters cmp3 and cmp4 respectively. The output of cmp4 matches the values of CMP_ERROR because there is no intervening functionality. For the filter cmp3, the *normal* (operational) *range* of the filter input is [-2, 2] which also happens to be a hard *range constraint* since there is a limit block before cmp3, constraining the range not to exceed [-2, 2] even if abnormal values are present at the input variable of the model named ERROR. Note that the range bounds reported in Figure 15 are convergent over time; i.e. filter output range is bounded by the values shown when input value varies arbitrarily within its range over infinite time. The range bounds of [-3.9506, 3.9506] for cmp3 output are precise and compliant with the theoretically expected bounds for the specific values of filter coefficients for cmp3. Likewise for cmp4.

The worst case numerical error bound (WCEB) for cmp3 output, after 72000 seconds of operation, is [-2.29e-004, -2.29e-004] over the range of [-3.9506, 3.9506] – this is two orders of magnitude less than the empirical practice tolerance of 1%.

4.2.2. Filter Example using a Variable Lag Filter

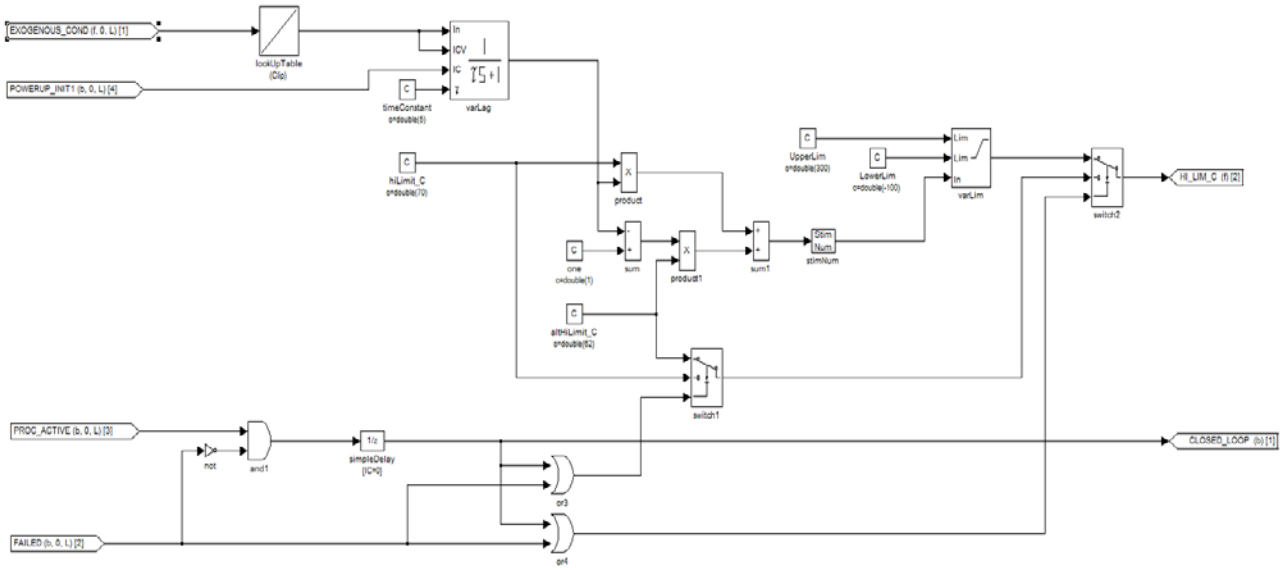

Figure 16 Variable Lag Filter - Test Model Diagram

Figure 16 shows 'high_lim_setpoint', a model containing a variable lag filter with a fixed tau. The next subsection describes the transfer function and difference equation derivation for this class of filters. Following subsection shows the analysis results assuming the filter runs for 72,000 seconds at 10 Hz (100 ms period).

4.2.2.1. Variable Lag Filter Transfer Function

Figure 17 shows the Variable Lag filter as it appears as a block in MATLAB Simulink. This class of filters has a variable tau, τ, that determines the lag according to the transfer function. There are additional *reset* (IC) and *reset value* (ICV) inputs that allow the filter to be reset.

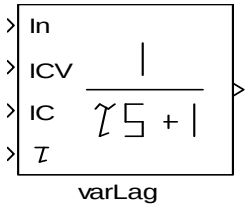

Figure 17 Variable Lag Filter

The following is the transfer function derivation for this filter class:

Reset Mode (when IC = 1): $\mathbf{y = ICV}$

Filter Mode Transfer Function (when IC = 0):

General form:

Continuous domain: $H(s) = \frac{1}{\tau s+1}$ **Discrete domain:** $H(z) = \frac{1+z^{-1}}{k_1+k_2 z^{-1}}$

Coefficient definitions (τ is given as an input)

$$k_1 = 1 + \frac{2\tau}{T_s} \qquad k_2 = 1 - \frac{2\tau}{T_s}$$

Difference equation:

$$y_n = (u_n + u_{n-1} - k_2 y_{n-1})/k_1 \tag{187}$$

4.2.2.2. Analysis results after 72,000 Seconds (20 hrs)

System Oper Time (secs): 72000 Model Period (ms): 100 Total Periods: 720000						
# Symbol Name	Normal Range Min	Normal Range Max	Range Constraint Min (if any)	Range Constraint Max (if any)	Data Type	Worst Case Numerical Error Bounds
CLOSED_LOOP	0	1			LangIndep_Bool	eB[-+0.00e+000]
HI_LIM_C	1.239987848	130.6000137	0	132	LangIndep_Float32	eB[-+4.53e-004]
# Filters						
varLag	0	0.980000196	0	1	LangIndep_Float32	eB[-+2.34e-007]

Figure 18 Variable Lag Filter - 72000 Seconds

In the above figure, CLOSED_LOOP is Boolean, and thus incurs no numerical error. HI_LIM_C shows the analysis for the final output of the model in Figure 16. Variable varLag shows the analysis for the output of the variable lag filter. The results are similar to that of the example in Section 4.2.1.2 and do not require further discussion.

4.2.3. Filter Example using a Lag Filter

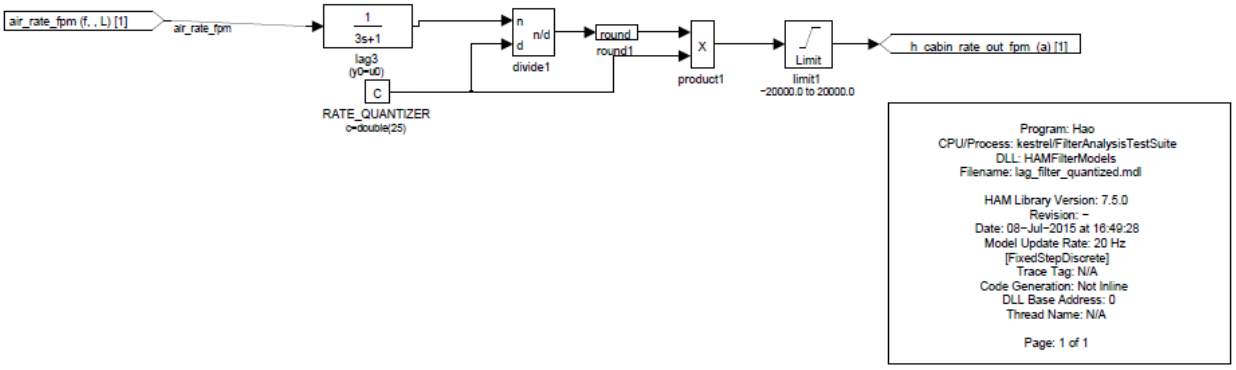

Figure 19 Lag Filter - Test Model Diagram

Figure 19 shows 'lag_filter_quantized', a model with a lag filter with a fixed time constant. The next subsection describes the transfer function and difference equation derivation for this class of filters; followed by the transfer function equations. Following subsection shows the analysis results assuming the filter runs for 72,000 seconds at 20 Hz (50 ms period).

4.2.3.1. Lag Filter Transfer Function

Figure 20 shows the Lag filter as it appears as a block in MATLAB Simulink. This class of filters has a time constant tau that is set for each filter instance. It is not resettable.

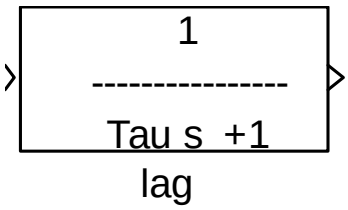

Figure 20 Lag Filter

The following is the transfer function derivation for this filter class:

Transfer Function:

General form:

Continuous domain: $H(s) = \frac{1}{\tau s+1}$ **Discrete domain:** $H(z) = \frac{1+z^{-1}}{k_1+k_2 z^{-1}}$

Coefficient definitions (all coefficients are constant)

$$k_1 = 1 + \frac{2\tau}{T_s} \qquad k_2 = 1 - \frac{2\tau}{T_s}$$

Difference equation:

$$y_n = (u_n + u_{n-1} - k_2 y_{n-1})/k_1 \tag{188}$$

4.2.3.2. *Analysis results after 72,000 Seconds (20 hrs)*

System Oper Time (secs): 72000 Model Period (ms): 50 Total Periods: 1440000						
# Symbol Name	Normal Range Min	Normal Range Max	Range Constraint Min (if any)	Range Constraint Max (if any)	Data Type	Worst Case Numerical Error Bounds
h_cabin_rate_out_fpm	0	225	-20000	20000	LangIndep_Float32	eB[-+0.00e+000]
# Filters						
lag3	11.999944	235.00006			LangIndep_Float32	eB[-+8.55e-010]

Figure 21 Lag Filter - 72000 Seconds

In the above figure, h_cabin_rate_out_fpm has numerical error bounds of zero because it is processed through a ‘round’ing block . The only other input affecting the value at h_cabin_rate_out_fpm is a whole number (the RATE_QUANTIZER constant of 25), which can be represented without loss. The values for lag3 above show the analysis as applicable to the output of the lag3 filter. The results are similar to that of the example in Section 4.2.1.2 and do not require further discussion.

4.2.4. Filter Example using a Quadratic Filter

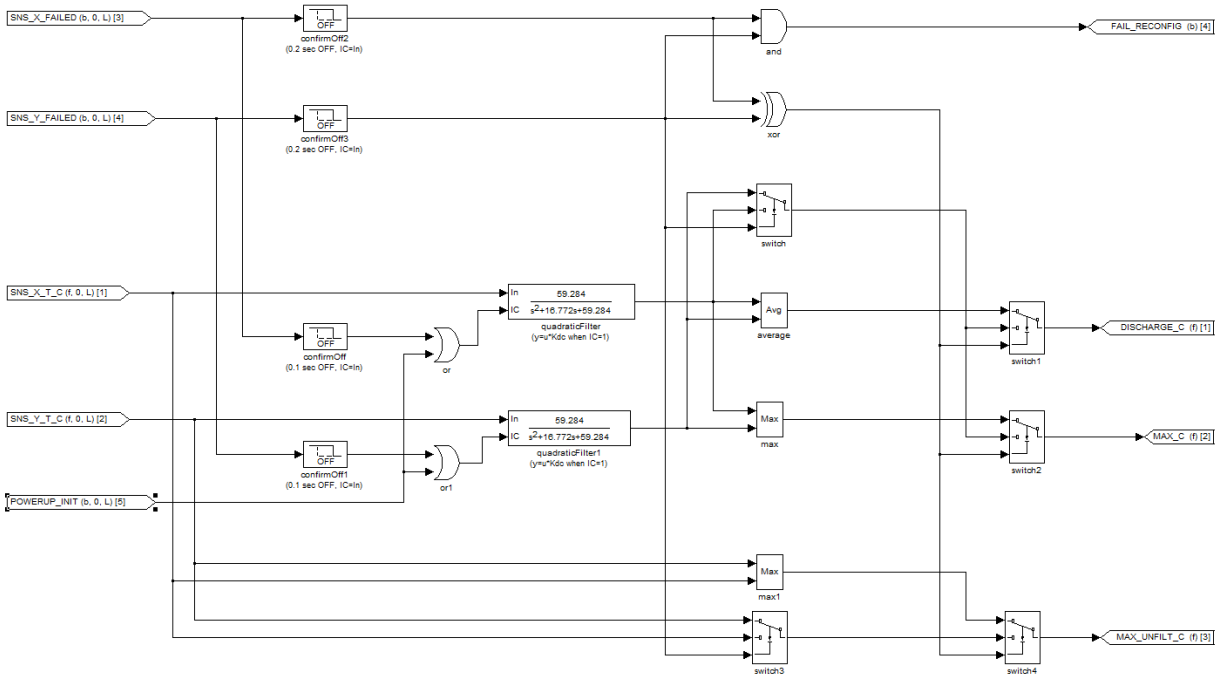

Figure 22 Quadratic Filter - Test Model Diagram

Figure 22 shows 'sensor_feedback_selection', a model with two quadratic filters. The next subsection describes the transfer function and difference equation derivation for this class of filters. Following subsection shows the analysis results assuming the filter runs for 72,000 seconds at 10 Hz (100 ms period).

4.2.4.1. Quadratic Filter Transfer Function

Figure 23 shows the Quadratic filter as it appears as a block in MATLAB Simulink. This class of filters has six constants specified for each instance: a,b,c, p,q,r that form the coefficients of two quadratic equations, a numerator: as^2 + bs+ c, and a denominator: ps^2+qs+r. There is also a *reset* (IC) that resets the filter to a pre-specified value.

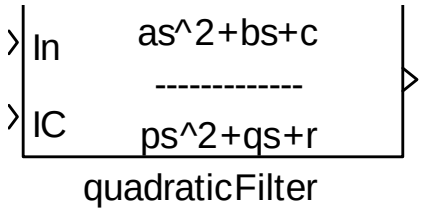

Figure 23 Quadratic Filter

The following is the transfer function derivation for this filter class:

Reset Mode (when $IC = 1$): $y = u * c/r$ **(c/r is the DC gain of the filter)**
Filter Mode Transfer Function (when $IC = 0$):

General form:

Continuous domain: $H(s) = \frac{as^2+bs+c}{ps^2+qs+r}$ **Discrete domain:** $H(z) = \frac{A+Bz^{-1}+Cz^{-2}}{D+Ez^{-1}+Fz^{-2}}$

Coefficient definitions (all coefficients are constant)

$$A = 4a + 2bT_s + cT_s^2 \qquad B = -2(4a - cT_s^2) \qquad C = 4a - 2bT_s + cT_s^2$$

$$D = 4p + 2qT_s + rT_s^2 \qquad E = -2(4p - rT_s^2) \qquad F = 4p - 2qT_s + rT_s^2$$

Difference equation:

$$y_n = (Au_n + Bu_{n-1} + Cu_{n-2} - Ey_{n-1} - Fy_{n-2})/D \tag{189}$$

4.2.4.2. Analysis results after 72,000 Seconds (20 hrs)

System Oper Time (secs): 72000 Model Period (ms): 100 Total Periods: 720000

# Symbol Name	Normal Range Min	Normal Range Max	Data Type	Worst Case Numerical Error Bounds
DISCHARGE_C	-49.99929177	32.00013311	LangIndep_Float32	eB[-+6.58e-003]
MAX_C	-49.99929177	32.00013311	LangIndep_Float32	eB[-+6.58e-003]
MAX_UNFILT_C	-50	32	LangIndep_Float32	eB[-+3.81e-006]
FAIL_RECONFIG	0	1	LangIndep_Bool	eB[-+0.00e+000]
# Filters				
quadraticFilter	-25.00008124	32.00013311	LangIndep_Float32	eB[-+3.12e-003]
quadraticFilter1	-49.99929177	-10.00002833	LangIndep_Float32	eB[-+6.58e-003]

Figure 24 Quadratic Filter - 72000 Seconds

The range bounds of the filter output variables in the above figure are compliant with the theoretically expected results, similar to that of the example in Section 4.2.1.2, and do not require further discussion.

In the above figure, FAIL_CONFIG has numerical error bounds of zero because it is a Boolean. DISCHARGE_C and MAX_C have the same numerical error bounds because they come from the results of the two quadratic filters. MAX_UNFILT_C has the error bound of a 32-bit floating point representation because it uses values straight from input without any processing. As you can see the error bounds on MAX_C and DISCHARGE_C are the max of the error bounds on the two quadratic filters, since both of them can end up with a value from either filter. It is remarkable that our static analysis techniques were able to prove numerical error bounds on the 2^{nd} order (quadratic) filter. The error bound of [-6.58e-003, 6.58e-003] on quadraticFilter1 output is within the practical tolerance guideline of 1% of the range. We conducted the analysis run for 72 seconds, 720 seconds, 7200 seconds, and 72000 seconds of operation. In all cases the error bounds is the same ([-6.58e-003, 6.58e-003]); implying it converges and stabilizes over time; thus supporting the theoretical analysis expectations.

4.2.5. Filter Example using a Variable Washout Filter

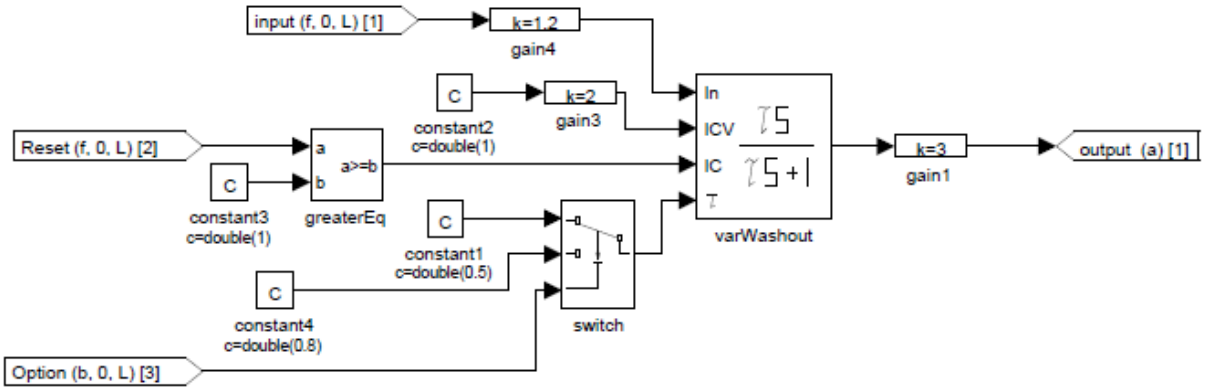

Figure 25 Variable Washout Filter - Test Model Diagram

Figure 25 shows 'VarWashoutFilter', a model with a variable washout filter. The next section (subsection 4.2.5.1) shows the transfer function diagram Figure 26 followed by the transfer function equations. Subsection 4.2.5.2 shows the analysis results assuming the filter runs for 72,000 seconds at 10 Hz (100 ms period). Subsection 4.2.5.3 shows the actual 'C' code for the filter.

4.2.5.1. Variable Washout Filter Transfer Function

Figure 26 shows the Variable Washout filter as it appears as a block in MATLAB Simulink. This class of filters has a variable tau, τ, that determines the lag according to the transfer function. There are additional *reset* (IC) and *reset value* (ICV) inputs that allow the filter to be reset.

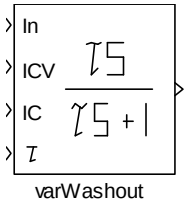

Figure 26 Variable Washout Filter

The following is the transfer function derivation for this filter class:

Reset Mode (when $IC = 1$): $y = ICV$

Filter Mode Transfer Function (when $IC = 0$):

General form:

Continuous domain: $H(s) = \frac{\tau s}{\tau s+1}$ **Discrete domain:** $H(z) = \frac{k_3+k_4 z^{-1}}{k_1+k_2 z^{-1}}$

Coefficient definitions (τ is given as an input)

$$k_1 = 1 + \frac{2\tau}{T_s} \qquad k_2 = 1 - \frac{2\tau}{T_s} \qquad k_3 = \frac{2\tau}{T_s} \qquad k_4 = -\frac{2\tau}{T_s}$$

Difference equation:

$$y_n = (k_3 u_n + k_4 u_{n-1} - k_2 y_{n-1})/k_1 \tag{190}$$

4.2.5.2. *Analysis results after 72,000 Seconds (20 hrs)*

System Oper Time (secs): 72000 Model Period (ms): 100 Total Periods: 720000				
# Symbol Name	**Normal Range Min**	**Normal Range Max**	**Data Type**	**Worst Case Numerical Error Bounds**
output	2811.069815	62188.08073	LangIndep_Float32	eB[-+1.11e+001]
# Filters				
varWashout	937.0232717	20729.36024	LangIndep_Float32	eB[-+3.71e+000]

Figure 27 Variable Washout Filter -72000 Seconds

In the above figure, you can see that output has an error bound of 3 times the error bound of varWashout, because they are separated by a gain of 3.

4.3. Summary of Accumulator Test Suite Results

The development of the second year analysis of Integrating Accumulators was guided by, tuned, and ultimately tested for round trip integration by a suite of test cases designed by the Honeywell team. These consisted of Simulink models of representative accumulator applications, their generated C implementations and their generated JSON files describing model-level properties of the generated code to be exploited by CodeHawk. This section summarizes the results of these test cases.

Motivation and Significance of the Results

The Patriot Missile system suffered an infamous failure [5] when its computation of time drifted over the course of two days of continuous operation. The computations using the 24-bit floating point unit introduced error in the calculation of elapsed time accumulated since system start. After 8 hours of operations, this resulted in a 20 percent error in the "range gate" – the expected position of the target. After 20 hours, the error was as much as 50 percent, resulting the Patriot missile's failure to intercept an incoming Scud missile. Figure 28 shows the increase in absolute error with elapsed time.

Hours	Seconds	Calculated Time (sec)	Inaccuracy (sec)	Approx. shift in Range Gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	8799.9725	.0251	55
20(a)	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100(b)	360000	359999.6667	.3433	687

Figure 28. The error in accumulated time due to floating point multiplication and the resulting distance by which the computed range gate was off

Using the combination of model-level and code-level analysis, we have addressed the problem of automated detection and analysis of such constructions. HiLiTE performance an analysis of cyclic path (feedback loop) invariants in the Simulink model to detect a counter type of invariants and provide this information to CodeHawk in terms of the accumulator variable and nature of the invariant. CodeHawk then applies novel code analysis techniques to derive precise closed-form mathematical bounds for the both the range and numerical error as a function of time. The results presented in the following subsections clearly show how the numerical error grows with system operational time.

4.3.1. Accumulator Example using a Fixed Integer Increment

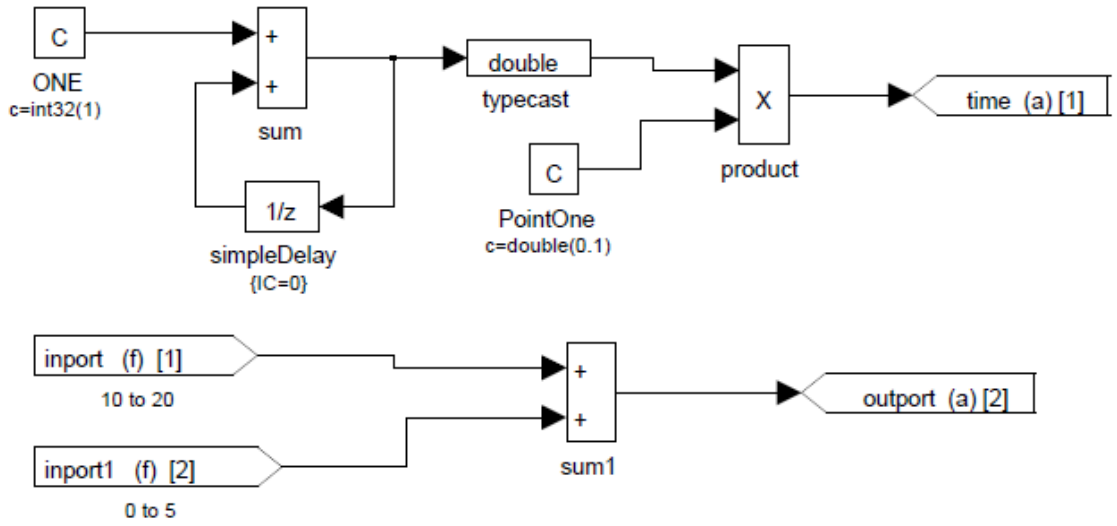

Figure 29. Fixed Increment Accumulator – abstract diagram for the Patriot Missile Bug

Figure 29 shows a model with an accumulator with a fixed integer increment which abstracts the essential elements of the Patriot Missile bug. At each periodic execution frame (at 10 Hz rate) the counter (output of sum block) counts by 1. The numerical error occurs when the counter value is multiplied by 0.1 to get the value of elapsed time; note that 0.1 is not accurately representable in any binary-based floating point notation. This numerical error is shown in the results.

The next subsection shows the analysis results assuming the filter runs for 72, 720, 7200 and 72,000 seconds at 10 Hz (100 ms period). The JSON files exchanged between HiLiTE and CodeHawk and the actual 'C' code.

4.3.1.1. Analysis results after different periods (72, 720, 7200, 72000 seconds)

System Oper Time (secs): 72 Model Period (ms): 100 Total Periods: 720
Accumulator pattern detected - Path Invariant Analysis:
Block : sum; Variable : y(k)_1; Invariant:
Guard: Assignment: y(k+1)_1=1+y(k)_1;

# Symbol Name	Normal Range Min	Normal Range Max	Data Type	Worst Case Numerical Error Bounds
time	0.1	72.00000107	LangIndep_Float32	eB[-+4.30e-005]
outport	10	25	LangIndep_Float32	eB[-+2.38e-006]

Figure 30 Fixed Increment Accumulator - 72 Seconds

System Oper Time (secs): 720 Model Period (ms): 100 Total Periods: 7200
Accumulator pattern detected - Path Invariant Analysis:
Block : sum; Variable : y(k)_1; Invariant:
Guard: Assignment: y(k+1)_1=1+y(k)_1;

#	Symbol Name	Normal Range Min	Normal Range Max	Data Type	Worst Case Numerical Error Bounds
time		0.1	720.0000107	LangIndep_Float32	eB[-+4.29e-004]
outport		10	25	LangIndep_Float32	eB[-+2.38e-006]

Figure 31 Fixed Increment Accumulator - 720 Seconds

System Oper Time (secs): 7200 Model Period (ms): 100 Total Periods: 72000
Accumulator pattern detected - Path Invariant Analysis:
Block : sum; Variable : y(k)_1; Invariant:
Guard: Assignment: y(k+1)_1=1+y(k)_1;

#	Symbol Name	Normal Range Min	Normal Range Max	Data Type	Worst Case Numerical Error Bounds
time		0.1	7200.000107	LangIndep_Float32	eB[-+4.31e-003]
outport		10	25	LangIndep_Float32	eB[-+2.38e-006]

Figure 32 Fixed Increment Accumulator - 7200 Seconds

System Oper Time (secs): 72000 Model Period (ms): 100 Total Periods: 720000
Accumulator pattern detected - Path Invariant Analysis:
Block : sum; Variable : y(k)_1; Invariant:
Guard: Assignment: y(k+1)_1=1+y(k)_1;

#	Symbol Name	Normal Range Min	Normal Range Max	Data Type	Worst Case Numerical Error Bounds
time		0.1	72000.00107	LangIndep_Float32	eB[-+4.48e-002]
outport		10	25	LangIndep_Float32	eB[-+2.38e-006]

Figure 33 Fixed Increment Accumulator - 72000 Seconds

In the above figure, the range and error bounds for the *time* variable are provide for different system operational times; both grow as a linear function of time as theoretically expected. (note that variable *outport* in this example is unrelated to the accumulator and thus its error bound remains fixed with the operational time).

4.3.2. Accumulator Example using Variable Increments

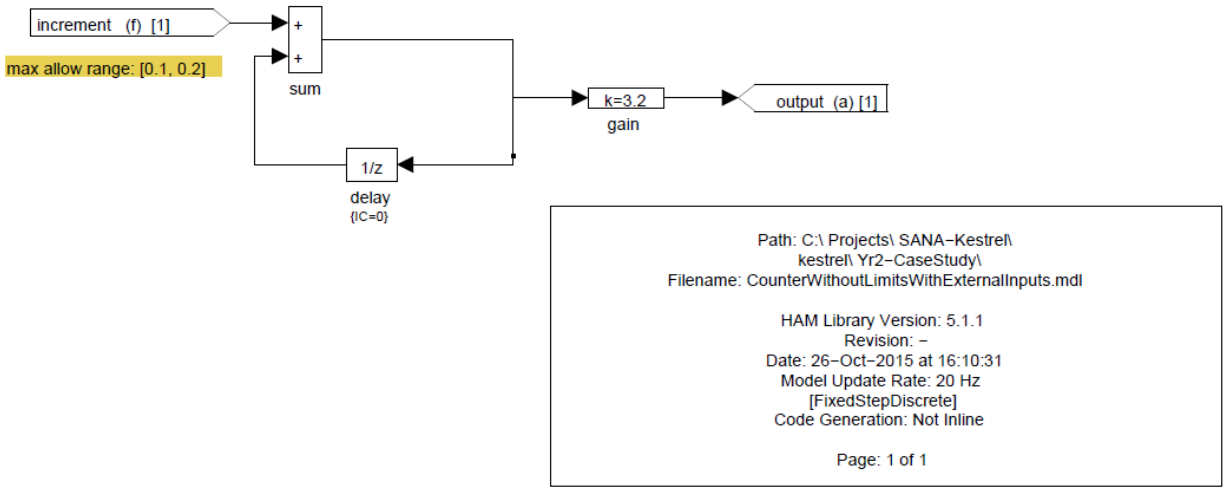

Figure 34 Variable Increment Accumulator - Test Model Diagram

The figure above shows 'CounterWithoutLimitsWithExternalInputs', a model with a floating-point accumulator with a variable increment with the input range constrained to [0.1, 0.2] . The next subsection shows the analysis results assuming the filter runs for 72, 720, 7200 72,000 seconds at 20 Hz (50 ms period). This example was created to ensure that both HiLiTE can CodeHawk can properly identify and analyze accumulators with floating-point variable increment. The results are consistent with the theoretical analysis.

4.3.2.1. Analysis results after different periods (72, 720, 7200, 72000 seconds)

System Oper Time (secs): 72 Model Period (ms): 50 Total Periods: 1440				
Accumulator pattern detected - Path Invariant Analysis:				
Block : sum; Variable : y(k)_1; Invariant:				
Guard: Assignment: y(k+1)_1=u(k+1)_0+y(k)_1;				
# Symbol Name	**Normal Range Min**	**Normal Range Max**	**Data Type**	**Worst Case Numerical Error Bounds**
output	0.32	921.5999794	LangIndep_Float32	eB[-+6.60e-004]
# Accumulators				
sum	0.1	288.0000043	LangIndep_Float32	eB[-+1.37e-004]

Figure 35 Variable Increment Accumulator - 72 Seconds

System Oper Time (secs): 720 Model Period (ms): 50 Total Periods: 14400				
Accumulator pattern detected - Path Invariant Analysis:				
Block : sum; Variable : y(k)_1; Invariant:				
Guard: Assignment: y(k+1)_1=u(k+1)_0+y(k)_1;				
# Symbol Name	**Normal Range Min**	**Normal Range Max**	**Data Type**	**Worst Case Numerical Error Bounds**
output	0.32	9215.999794	LangIndep_Float32	eB[-+6.60e-003]
# Accumulators				
sum	0.1	2880.000043	LangIndep_Float32	eB[-+1.37e-003]

Figure 36 Variable Increment Accumulator - 720 Seconds

System Oper Time (secs): 7200 Model Period (ms): 50 Total Periods: 144000				
Accumulator pattern detected - Path Invariant Analysis:				
Block : sum; Variable : y(k)_1; Invariant:				
Guard: Assignment: y(k+1)_1=u(k+1)_0+y(k)_1;				
# Symbol Name	**Normal Range Min**	**Normal Range Max**	**Data Type**	**Worst Case Numerical Error Bounds**
output	0.32	92159.99794	LangIndep_Float32	eB[-+6.65e-002]
# Accumulators				
sum	0.1	28800.00043	LangIndep_Float32	eB[-+1.39e-002]

Figure 37 Variable Increment Accumulator - 7200 Seconds

System Oper Time (secs): 72000 Model Period (ms): 50 Total Periods: 1440000				
Accumulator pattern detected - Path Invariant Analysis:				
Block : sum; Variable : y(k)_1; Invariant:				
Guard: Assignment: y(k+1)_1=u(k+1)_0+y(k)_1;				
# Symbol Name	**Normal Range Min**	**Normal Range Max**	**Data Type**	**Worst Case Numerical Error Bounds**
output	0.32	921599.9794	LangIndep_Float32	eB[-+7.19e-001]
# Accumulators				
sum	0.1	288000.0043	LangIndep_Float32	eB[-+1.50e-001]

Figure 38 Variable Increment Accumulator - 72000 Seconds

5. CONCLUSIONS

We have demonstrated a combined model-level and code-level analysis approach to prove precise, yet conservative bounds for variable ranges and numerical error on two classes of numerical algorithms: linear digital filters and accumulators. Our novel analysis approach uses the semantics of control algorithms available in the Simulink models to identify and classify the properties of filter blocks and accumulator patterns. The HiLiTE tool introduced a novel technique to analyze cyclic path (feedback loop) expressions in models to automatically detect patterns and reduce them to temporal invariants in terms of a single accumulator variable that could then be provided to CodeHawk. We defined extensible formats for representing variable ranges, properties of filters and accumulators, functions for error bounds. These were used in developing a flexible JSON interchange infrastructure in the tool chain that allowed the automated round-trip analysis and presentation of the results to the user.

At the code-level analysis, Anca Browne discovered a technique, based on classical discrete mathematics, for computing the exact (real) bounds on linear discrete filters, and applied it to first-order and second-order filters supplied by Honeywell. The time to analyze a filter is negligible. This contrasts with analyses performed for Airbus that take on the order of 20 hours to get less precise results.

Extending this technique to floating point error analysis, she discovered how to generate a closed-form solution for the error as a function of the number of iterations of the recurrence that are performed. Again, this result was applied to first-order and second-order filters supplied by Honeywell. The result of the error analysis were sufficiently precise and valid even for second-order filters the numerical error stabilized within practically acceptable tolerance bounds even for very large periods of run time. These precise bounds prove the essential characteristics of

variable ranges and numerical error in the control algorithms, as discussed in detail in Section 4.2.

These techniques were slightly generalized to analyze a different class of numerical algorithms used in control systems, namely accumulators. Accumulators, such as counters, are used in many control systems – a prime example is the Patriot Missile system where the accumulated numerical error resulted in an infamous failure. Again, our techniques allow almost instantaneous generation of an analysis function that gives (1) accumulator results as a function of the number of iterations performed, and (2) accumulated floating-point error as a function of the number of iterations performed.

As avionics software becomes increasingly complex, the need for automated analysis becomes critical to verify correct algorithm behavior and absence of potential failures. The current static analysis tools and proposed techniques may handle discrete types of algorithms well, but fall short of providing a precise analysis for numerically intensive algorithms. This can result in large range bounds (or no error bounds) provided by the tools which then leave it to the developer to analyze the problem by manual reviews or empirical means. This is already proving to be a problem in current avionics system when deploying large amounts of software with frequent configuration changes. For example, developers may need to do extensive analysis to ascertain that the new set of values of filter parameters is not degenerate and will indeed make the range and error bounds converge.

We believe this work is a significant step towards improving the automation and quality of results provided by static analysis techniques. Our novel analysis techniques can automatically analyze the code implementations of a variety of first and second order linear filter and derive not only precise range bounds but also numerical error bounds. This allows quick verification of normal algorithm behavior as well as detection of erroneous combinations filter parameters (time constant values) that can make bounds diverge, and defective implementations that can give rise to large numerical error. Our techniques for the analysis of accumulator patterns would have detected the bug that led to Patriot Missile failure, and may hopefully prevent such faulty software in the future.

6. REFERENCES

[1] Patrick Cousot and Radhia Cousot, Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, POPL, 1977, pp 238-252.

[2] Cousot, Patrick, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTRÉE analyzer." In *Programming Languages and Systems*, pp. 21-30. Springer Berlin Heidelberg, 2005.

[3] Feret, Jérôme. "Static analysis of digital filters." In *Programming Languages and Systems*, pp. 33-48. Springer Berlin Heidelberg, 2004.

[4] K. Eriksson, A Summary of Recursion Solving Techniques, KTH technical note, https://www.math.kth.se/math/GRU/2012.2013/SF1610/CINTE/mastertheorem.pdf .

[5] The Patriot Missile Failure, http://www.ima.umn.edu/~arnold//disasters/patriot.html.

7. LIST OF ACRONYMS

COM	**Common Object Model**
HiLiTE	**Honeywell Integrated Lifecycle Tool Environment**
JSON	**JavaScript Object Notation**
LDF	**Linear Digital Filters**
MBD	**Model-Based Development**
Ocaml	**O Collaborative Application Markup Language**
PI	**Proportional-Integrator**
SANA	**Static Analysis of Numerical Algorithms**
SoW	**Statement of Work**
WCEB	**Worst Case Error Bound**